Semantic subtyping: dealing set-theoretically with union, intersection, and negation types

Alain Frisch
INRIA Roquencourt

Giuseppe Castagna
École Normale Supérieure de Paris

Véronique Benzaken
LRI - Université Paris Sud

Abstract

Subtyping relations are usually defined either syntactically by a formal system or semantically by an interpretation of types into an untyped denotational model. This work shows how to define a subtyping relation semantically in the presence of Boolean connectives, functional types and dynamic dispatch on types, without the complexity of denotational models, and how to derive a complete subtyping algorithm.

1 Introduction

Many recent type systems rely on a subtyping relation. Its definition generally depends on the type algebra, and on its intended use. We can distinguish two main approaches for defining subtyping: the *syntactic* approach and the *semantic* one. The syntactic approach—by far the more used—consists in defining the subtyping relation by axiomatising it in a formal deduction system (a set of inductive or co-inductive rules); in the semantic approach (for instance, [1, 12]), instead, one starts with a model of the language and an interpretation of types as subsets of the model, then defines the subtyping relation as the inclusion of denoted sets, and, finally, when the relation is decidable, derives a subtyping algorithm from the semantic definition.

The semantic approach has several advantages but it is also more constraining. Finding an interpretation in which types can be interpreted as subsets of a model may be a hard task. A solution to this problem was given by Haruo Hosoya and Benjamin Pierce [20, 18, 21] with the work on XDuce. The key idea is that in order to define the subtyping relation semantically one does not need to start from a model of the whole language: a model of the types suffices. In particular Hosoya and Pierce take as model of types the set of values of the language. Their notion of model cannot capture functional values. On the one hand, the resulting type system is poor since it lacks function types. On the other hand, it manages to integrate union, product and recursive types and still

keep the presentation of the subtyping relation and of the whole type system quite simple.

In a previous work [16, 14] we extended the work on XDuce and re-framed it in a more general setting: we show a technique to define semantic subtyping in the presence of a rich type system including function types, but also arbitrary Boolean combinations (union, intersection, and negation types) and in the presence of lately bound overloaded functions and type-based pattern matching. The aim of [16, 14] was to provide a theoretical foundation on the top of which to build the language ℂDuce [4], an XML-oriented transformation language. The key theoretical contribution of the work is a new approach to define semantic subtyping when straightforward set-theoretic interpretation does not work, in particular for arrow types. Here we focus and expand on this aspect of the work and we get rid of many features (e.g. pattern matching and pattern variable type inference) which are not directly related to the treatment of subtyping.

The description of a general technique to extend semantic subtyping to general types systems with arrow and complete Boolean combinator types is just one way to read our work, and it is the one we decided to emphasise in this presentation. However it is worth mentioning that there exist at least two other readings for the results and techniques presented here.

A first alternative reading is to consider this work as a research on the definition of a general purpose higher-order XML transformation language: indeed, this was the initial motivation of [16, 14] and the theoretical work done there constitutes the fundamental basis for the definition *and the implementation* of the XML transformation language ℂDuce.

A second way of understanding this work is as a quest for the generalisation of lately bound overloaded functions to intersections types. The intuition that overloaded functions should be typed by intersection types was always felt but never fully formalised or understood. On the one hand we had the long-standing research on intersection types with the seminal works by the Turin research group on typed lambda calculus [3, 11]. However functions with intersection types had a uniform behaviour, in the sense that even if they worked on arguments of different types they always executed the same code on all of these types[1]. So functions with intersections types looked closer to parametric polymorphism (in which we enumerate the possible domains) rather than overloaded functions which are able to discriminate on the type of the argument and execute a different code for each different type. On the other hand there was the research on overloaded functions as used in programming languages which accounted for functions formed by different pieces of code selected according to the type of the argument the function is applied to. However, even if the types of these functions are apparently close to intersection types, they never had the set theoretic intuition of intersections. So for example in the $\lambda\&$-calculus [8] overloaded functions have types that are characterised by the same subtyping relation as intersection types, but they differ from the latter by the need of spe-

[1] A notable exception to this is John Reynolds work on the coherent overloading and the language Forsythe [22, 23].

cial formation rules that have no reasonable counterpart in intersection types. The overloaded functions defined here and, even more, those defined in [16] finally reconcile the two approaches: they are typed by intersection types (with a classical/set-theoretic interpretation) and their definitions may intermingle code shared by all possible input types with pieces of code that are specific to only some particular input types. Therefore they nicely integrate the two styles of programming.

Finally it is important to stress that although here we deploy our construction for a λ-calculus with higher-order functions, the technique is quite general and can be used mostly unchanged for quite different paradigms, as for instance it is done in [9] for the π-calculus.

Plan of the article. The presentation is structured in three parts:

1. In the first part (Section 2) we lengthy discuss the main ideas, the underlying intuitions, and the logical entailment of the whole approach.
2. In the second part (Sections 3–5) we succinctly and precisely define the system: the calculus and its typing relation (Section 3), the subtyping relation (Section 4), and their properties (Section 5).
3. The last part (Section 6) presents the technical details of the properties stated in the second part.

Section 7 concludes our presentation.

2 Overview of the approach

When dealing with syntactic subtyping one usually proceeds as follows. First, one defines a language, then, somewhat independently, the set of (syntactic) types and a subtyping relation on this set. This relation is defined axiomatically, in an inductive (or co-inductive, in case of recursive types) way. The type system, consisting of the set of types and of the subtyping relation, is coupled to the language by a *typing relation*, usually defined via some typing rules by induction on the terms of the language and possibly a *subsumption* rule that accounts for subtyping. The meaning of types is only given by the rules defining the subtyping and the typing relations.

The semantic subtyping approach described here diverges from the above only for the definition of the subtyping relation. Instead of using a set of deduction rules, this relation is defined semantically: we do it by defining a *set-theoretic* model of the types and by stating that one type is subtype of another if the interpretation of the former is a *subset* of the interpretation of the latter. As for syntactic subtyping, the definition is parametric in the set of base types and their subtyping relation (in our case, their interpretation).

2.1 A five steps recipe

In principle, the process of defining semantic subtyping can be roughly summarised in the following five steps:

1. Take a bunch of type *constructors* (e.g., $\rightarrow$, $\times$, $\mathtt{ch}$, ...) and extend the type algebra with the following *Boolean combinators*: union $\vee$, intersection $\wedge$, and negation $\neg$, yielding a type algebra $\mathscr{T}$.

2. Give a *set-theoretic model* of the type algebra, namely define a function $[\![\,]\!]_D : \mathscr{T} \rightarrow \mathscr{P}(D)$, for some domain D (where $\mathscr{P}(D)$ denotes the powerset of D). In such a model, the combinators must be interpreted in a set-theoretic way (that is, $[\![s \wedge t]\!]_D = [\![s]\!]_D \cap [\![t]\!]_D$, $[\![s \vee t]\!]_D = [\![s]\!]_D \cup [\![t]\!]_D$, and $[\![\neg t]\!]_D = D \setminus [\![t]\!]_D$), and the definition of the model must capture the essence of the type constructors.

   There might be several models, and each of them induces a specific subtyping relation on the type algebra. We only need to prove that there exists at least one model and then pick one that we call the *bootstrap model*. If its associated interpretation function is $[\![\,]\!]_{\mathscr{B}}$, then it induces the following subtyping relation:

$$s \leq_{\mathscr{B}} t \quad \stackrel{def}{\Longleftrightarrow} \quad [\![s]\!]_{\mathscr{B}} \subseteq [\![t]\!]_{\mathscr{B}} \tag{1}$$

3. Now that we defined a subtyping relation for our types, find a subtyping algorithm that decides (or semi-decides) the relation. This step is not mandatory but highly advisable if we want to use our types in practice.

4. Now that we have a (hopefully) suitable subtyping relation available, we can focus on the language itself, consider its typing rules, use the new subtyping relation to type the terms of the language, and deduce $\Gamma \vdash_{\mathscr{B}} e : t$. In particular this means to use in the subsumption rule the bootstrap subtyping relation $\leq_{\mathscr{B}}$ we defined in step 2.

5. The typing judgement for the language now allows us to define a *new* natural set-theoretic interpretation of types, the one based on values $[\![t]\!]_{\mathscr{V}} = \{v \in \mathscr{V} \mid \vdash_{\mathscr{B}} v : t\}$, and then define a "new" subtyping relation as we did in (1), namely $s \leq_{\mathscr{V}} t \stackrel{def}{\Leftrightarrow} [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$. The new relation $\leq_{\mathscr{V}}$ might be different from $\leq_{\mathscr{B}}$ we started from. However, if the definitions of the model, of the language, and of the typing rules have been carefully chosen, then the two subtyping relations coincide

$$s \leq_{\mathscr{B}} t \quad \Longleftrightarrow \quad s \leq_{\mathscr{V}} t$$

   and this closes the circularity. Then, the rest of the story is standard (reduction relation, subject reduction, type-checking algorithm, etc ...).

While the five steps above outline a nice framework in which to fit and understand what follows, in practice, however, the starting point never is the model of types but the calculus: in particular one always starts from the calculus and its

values, and tries to slightly modify these so that the values outline some model that can then be formalised. This is what we also do here: while we follow the five-steps processes above to give, in the rest of this section, an overview of the approach, in Section 3 we introduce a λ-calculus with overloaded functions and dynamic dispatch, in Section 4 we introduce a model to semantically define a subtyping relation inspired from the previous calculus, and in Section 5 discuss the main results, namely, the soundness of the typing relation, the correspondence between the values of Section 3 and the model of Section 4, and the decidability of the various relations.

2.2 Advantages of semantic subtyping

The semantic approach is more technical and constraining, and this may explain why it has obtained less attention than syntactic subtyping. However it presents several advantages:

1. When type constructors have a natural interpretation in the model, the subtyping relation is by definition complete with respect to its intuitive interpretation as set inclusion: when $t \leq s$ does not hold, it is possible to exhibit an element of the model which is in the interpretation of t and not of s, even in presence of arrow types (this property is used in $\mathbb{C}$Duce to return informative error messages to the programmer); in the syntactic approach one can just say that the formal system does not prove $t \leq s$, and there may be no clear criterion to assert that some meaningful additional rules would not allow the system to prove it. This argument is particularly important with a rich type algebra, where type constructors interact in non trivial ways; for instance, when considering arrow, intersection and union types, one must take into account —i.e., introduce rules for— many distributivity relations such as, for instance[2], $(t_1 \vee t_2) \to s \simeq (t_1 \to s) \wedge (t_2 \to s)$. Forgetting any of these rules yields a type system that, although sound, does not match (that is, it is not complete with respect to) the intuitive semantics of types.

2. In the syntactic approach deriving a subtyping algorithm requires a strong intuition of the relation defined by the formal system, while in the semantic approach it is a simple matter of "arithmetic": it simply suffices to use the interpretation of types and well-know Boolean algebra laws to decompose subtyping on simpler types (as we show in Section 6.2). Furthermore, as most of the formal effort is done with the semantic definition of subtyping, studying variations of the algorithm (e.g., optimisations or different rules) turns out to be much simpler (this is common practise in database theory where, for example, optimisations are derived directly from the algebraic model of data).

3. While the syntactic approach requires tedious and error-prone proofs of formal properties, in the semantic approach many of them come for free:

[2]We write $s \simeq t$ as a shorthand for $s \leq t$ *and* $s \geq t$.

for instance, the transitivity of the subtyping relation is trivial (as set-containment is transitive), and this makes proofs such as cut elimination or transitivity admissibility pointless. Other examples of properties that come easily from a semantic definition are the variance of type constructors, and distributivity laws (e.g. $t_1 \times (t_2 \vee t_3) \simeq (t_1 \times t_2) \vee (t_1 \times t_3)$).

Although these properties look quite appealing, the technical details of the approach hinder its development: in the semantic approach, one must be very careful not to introduce any circularity in the definitions. For instance, if the type system depends on the subtyping relation—as this is generally the case—one cannot use it to define the semantic interpretation which must thus be untyped; also, usually the model corresponds to an untyped denotational semantics, and types are interpreted as ideals and this precludes the set-theoretic interpretation of negative types (as the complement of ideals is not an ideal). For these reasons all the semantic approaches to subtyping previous to our work presented some limitations: no higher-order functions, no complement types, and so on. The main contribution of our work is the development of a formal framework that overcomes these limitations.

EXCURSUS. The reader should not confuse our research with the long-standing research on set-theoretic models of subtyping. In that case one starts from a syntactically (i.e. axiomatically) defined subtyping relation and seeks a set-theoretic model where this relation is interpreted as inclusion. Our approach is the opposite: instead of starting from a subtyping relation to arrive to a model, we start by defining a model in order to arrive to a subtyping relation. Thus in our approach types have a strong substance even before introducing the typing relation.

2.3 A model of types

To define semantic subtyping we need a set-theoretic model of types. The source of most of (if not all) the problems comes from the fact that this model is usually defined by starting from a model of the terms of the language. That is, we consider a denotational interpretation function that maps each term of the language into an element of a semantic domain and we use this interpretation to define the interpretation of the types (typically—but not necessary, e.g. PER models [2]—as the image of the interpretation of all terms of a given type). If we consider functional types then in order to interpret functional term application we have to interpret the duality of functions as terms and as functions on terms. This yields the need to solve complicated recursive domain equations that hardly combines with a set-theoretic interpretation of types, whence the introduction of restrictions in the definition of semantic subtyping (e.g. no function types, no negation types, etc ...).

Note however that in order to define semantic subtyping all we need is a set-theoretic *model of types*. The construction works even if we do not have a

model of terms. To push it to the extreme, in order to define subtyping we do not need terms at all, since we could imagine to define type inclusion for types independently from the language we want to use these types for. More plainly, the definition of a semantic subtyping relation needs neither an interpretation for applications (that is an applicative model) nor, thus, the solution of complicated domain equations.

The key idea to generalise semantic subtyping is then to dissociate the *model of types* from the *model of terms* and define the former independently from the latter. In other words, the interpretation of types must not forcedly be based on, or related to an interpretation of terms (and actually in the some concrete examples we will give we interpret types in structures that cannot be used for an interpretation of terms), and as a matter of fact we do not need an interpretation of terms even to exist for the semantic subtyping construction to go through[3].

2.4 Types as sets of values

Nevertheless, to ensure type safety (i.e. well-typed programs cannot go wrong) the meaning of types has to be somewhat correlated with the language. A classical solution, that belongs to the types folklore[4] is to interpret types as sets of *values*, that is, as the results of *well-typed* computations in the language. More formally, the values of a typed language are all the terms that are well-typed, closed, and in normal form. So the idea is that in order to provide an interpretation of types we do not need an interpretation of all terms of the language (or of just the well-typed ones): the interpretation of the values of the language suffices to define an interpretation of types. This is much an easier task: since a closed application usually denotes a redex, then by restricting to the sole values we avoid the need to interpret application and, therefore, also the need to solve complicated domain equations. This is the solution adopted by XDuce, where values are XML documents and types are sets of documents (more precisely, regular languages of documents).

But if we consider a language with arrow types, that is a language with higher order functions, then the applications come back again: arrow types must be interpreted as sets of function values, that is, as sets of well-typed closed lambda abstractions, and applications may occur in the body of these abstractions. Here is where XDuce stops and it is the reason why it does not include arrow types.

[3] As Pierre-Louis Curien suggested, the construction we propose is a *pied de nez* to (it cocks a snook at) denotational semantics, as it uses a semantic construction to define a language for which, possibly, no denotational semantics is known.

[4] A survey on the "Types" mailing list traces this solution back to Bertrand Russell and Alfred Whitehead's Principia Mathematica. Closer to our interests it seems that the idea independently appeared in the late sixties early seventies and later back again in seminal works by Roger Hindley, Per Martin-Löf, Ed Lowry, John Reynolds, Niklaus Wirth and probably others (many thanks to the many "typers" who answered to our survey).

2.5 A circularity to break

Introducing arrow types is then problematic because it slips applications back again in the interpretation of types. However this does not mean that we need a semantic interpretation for application, it just implies that we must define how application is *typed*. Indeed, functional values are *well-typed* lambda abstractions, so to interpret functional types we must be able to type lambda abstractions and in particular to type the applications that occur in their body. Now this is not an easy task in our context: in the absence of higher order functions the set of values of type constructors such as products or records can be inductively defined from basic types without resorting to any typing relation (this is why the XDuce approach works smoothly). With the arrow type constructor, instead, this can be done only by using a typing relation, and this yields to the circularity we hinted at in the introduction and that is shown in Figure 1: in order to define the subtyping relation we need an interpretation of the types of the language; for this we have to define which are the values of an arrow type; this needs that we define the typing relation for applications, which in turns needs the definition of the subtyping relation.

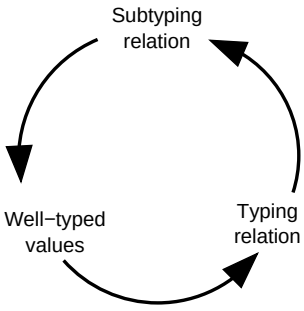

Figure 1: Circularity

Thus, if we want to define the semantic subtyping of arrow types we must find a way the avoid this circularity. The simplest way to avoid it is to break it, and the development we did so far clearly suggests where to break it. We always said that to define (semantic) subtyping we *must* have a model of types; it is also clear that the typing relation *must* use subtyping; on the contrary it is not strictly necessary for our model to be based on the interpretation of values, this is just convenient as it ties the types with the language the types are intended for. This is therefore the weakest link and we can break it. So the idea is to start from a model (of the types) defined independently (but not too much) from the language the types are intended for (and therefore independently from its values), and then from that define the rest: subtyping, typing, set of values. We will then show how to relate the initial model to the obtained language and recover the initial "types as set of values" interpretation: namely, we will "close the circle".

2.6 Set-theoretic models

Let us then show more in details how we shall proceed. We do not need to define a particular language, the definition of types will suffice. Here, we assume that types are defined by the following syntax:

$$t ::= \mathbb{0} \mid \mathbb{1} \mid t \to t \mid t \times t \mid \neg t \mid t \vee t \mid t \wedge t$$

where $\mathbb{0}$ and $\mathbb{1}$ respectively correspond to the empty and universal types (these are sometimes denoted by the pair $\bot$, $\top$ or **Bottom**, **Top**). The formal defini-

tion of the type algebra, which includes recursive types and basic types, will be given in Section 3.1.

The second step is to define precisely what a *set-theoretic* model for these types is. As Hindley and Longo [17] give some general conditions that characterise models of λ-calculus, so here we want to give the conditions that an interpretation function must satisfy in order to characterise a set-theoretic model of our types. So let $\mathscr{T}$ be the set of types, D some set, and $[\![\_]\!]$ an interpretation function from $\mathscr{T}$ to $\mathscr{P}(D)$. The conditions that $[\![\_]\!]$ must satisfy to define a set-theoretic model are mostly straightforward, namely:

1. $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$
2. $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$
3. $[\![\neg t]\!] = D \backslash [\![t]\!]$
4. $[\![\mathbb{1}]\!] = D$
5. $[\![\mathbb{0}]\!] = \varnothing$
6. $[\![t \times s]\!] = [\![t]\!] \times [\![s]\!]$
7*. $[\![t \to s]\!] =$???

The first six conditions convey the intuition that our model is set theoretic: so the intersection of types must be interpreted as set intersection, the union of types as set-theoretic union and so on (the sixth condition requires some closure properties on D but we prefer not to enter in such a level of detail at this point of our presentation). But the definition is not complete yet as we still have to establish the seventh condition (highlighted by a $*$) that constrains the interpretation of arrow types. This condition is more complicated. Again it must convey the intuition that the interpretation is set theoretic, but while the first six conditions are language independent, this conditions strongly depends on the language and in particular on the kind of functions we want to implement in our language. We give detailed examples about this in [14]. The set theoretic intuition we have of function spaces is that a function is of type $t \to s$ if whenever applied to a value of type t it returns a result of type s. Intuitively, if we interpret functions as binary relations on D, then $[\![t \to s]\!]$ is the set of binary relations in which if the first projection is in (the interpretation of) t then the second projection is in s, namely $\{ f \subseteq D^2 \mid \forall (d_1, d_2) \in f.\ d_1 \in [\![t]\!] \Rightarrow d_2 \in [\![s]\!] \}$. Note that this set can also be written $\mathscr{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$, where the overline denotes set complement. If the language is expressive enough, we can do as if every binary relation in this set was an element of $[\![t \to s]\!]$; thus, we would like to say that the seventh condition is:

$$[\![t \to s]\!] = \mathscr{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}}) \tag{2}$$

But this is completely meaningless. First, technically, this would imply that $\mathscr{P}(D^2) \subseteq D$, which is impossible for cardinality reasons. Also, remember that

we want eventually to re-interpret types as sets of values of the language, and functions in the language are *not* binary relations (they are syntactic objects). However what really matters is not the exact mathematical nature of the elements of D, but only the relations they create between types. The idea then is to do *as if* the above condition held.

Since this point is central to our model, let us explain it differently. Recall that the only reason why we want to accurately state what set-theoretic model of types is, is to precisely define the subtyping relation for syntactic types. In other words, we do not define an interpretation of types in order to formally and mathematically state what the syntactic types *mean* but, more simply, we define it in order to state how they are *related*. So, even if we would like to say that a type $t\!\rightarrow\!s$ must be interpreted in the model as $\mathscr{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$ as stated by (2), for what it concerns the goal we are aiming at, it is enough to require that a model must interpret functional types so as the induced subtyping relation is the same as the one the condition (2) would induce, that is:

$$[\![t_1\!\rightarrow\!s_1]\!] \subseteq [\![t_2\!\rightarrow\!s_2]\!] \iff \mathscr{P}(\overline{[\![t_1]\!] \times \overline{[\![s_1]\!]}}) \subseteq \mathscr{P}(\overline{[\![t_2]\!] \times \overline{[\![s_2]\!]}})$$

and similarly for any Boolean combination of arrow types.

Formally, we associate (see Definition 4 in Section 4.2) to $[\![\_]\!]$ an extensional interpretation $\mathbb{E}(\_)$ that behaves as $[\![\_]\!]$ except for arrow types, for which we use the condition above as definition:

$$\mathbb{E}(t\!\rightarrow\!s) = \mathscr{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$$

Note that we use $[\![\_]\!]$ in the right-hand side of this equation, that is, we only re-interpret top-level arrow types. Now we can express the fact that $[\![\_]\!]$ behaves (from the point of view of subtyping) as if functions were binary relations. This is obtained by writing the missing seventh condition, not in the form of 7^*, but as follows:

7. $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$

or, equivalently, $[\![t_1]\!] \subseteq [\![t_2]\!] \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$.[5]

To put it otherwise, if we wanted an interpretation $[\![\_]\!]$ of the types that were faithful with respect to the semantics of the language, then we should require for all t that $[\![t]\!] = \mathbb{E}(t)$. But for cardinality reasons this is impossible in a set-theoretic framework. However we do not need such a strong constraint on the definition of $[\![\_]\!]$ since all we ask to $[\![\_]\!]$ is to characterise the *containment* of types, and to that end it suffices to characterise the zeros of $[\![\_]\!]$, since

$$s \leq t \iff [\![s]\!] \subseteq [\![t]\!] \iff [\![s]\!] \cap \overline{[\![t]\!]} = \varnothing \iff [\![s\wedge\neg t]\!] = \varnothing$$

Therefore, instead of asking that $[\![\_]\!]$ and $\mathbb{E}(\_)$ coincide on all points, we require a weaker constraint, namely that they have the same zeros:

$$[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$$

[5] Indeed, $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \setminus [\![t_2]\!] = \varnothing \iff [\![t_1\wedge\neg t_2]\!] = \varnothing \iff \mathbb{E}(t_1\wedge\neg t_2) = \varnothing \iff \mathbb{E}(t_1) \setminus \mathbb{E}(t_2) = \varnothing \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$.

This is the essence of our definition of *models* of the type algebra (Definition 5 in Section 4.2).

We said that the above seventh condition (actually, the definition of the extensional interpretation) depends on the language the type system is intended for. Previous work [14] shows different variations of this conditions to match different sets of definable transformations. However, we can already see that the condition above accounts for languages in which functions possibly are

1. *Non-deterministic*: since the condition does not prevent the interpretation of a function space to contain a relation with two pairs (d, d_1) and (d, d_2) with $d_1 \neq d_2$.

2. *Non-terminating*: since the condition does not force a relation in $[\![t{\to}s]\!]$ to have as first projection the whole $[\![t]\!]$. A different reason for this is that every arrow type is inhabited (note indeed that the empty set belongs to the interpretation of every arrow type), so in particular are all the types of the form $t{\to}\mathbb{0}$; now, all the functions in such types must be always non-terminating on their domain (if they returned a value this would inhabit $\mathbb{0}$).

3. *Overloaded*: this is subtler than the two previous cases as it is a consequence of the fact that condition does not force $[\![(t_1{\vee}t_2){\to}(s_1{\wedge}s_2)]\!]$ to be equal to $[\![(t_1{\to}s_1){\wedge}(t_2{\to}s_2)]\!]$, but just the former to be included in the latter. Imagine indeed that the language at issue does not allow the programmer to define overloaded functions. So it may be not possible to define functions that distinguish the types of their argument, and in particular to have a function that when applied to an argument of type t_1 returns a result in s_1 while returns a (possibly different) s_2 result for t_2 arguments. Therefore the only functions in $(t_1{\to}s_1){\wedge}(t_2{\to}s_2)$ are those in $(t_1{\vee}t_2){\to}(s_1{\wedge}s_2)$ (this point is discussed thoroughly in Section 4.5 of our related survey [5]).

2.7 Bootstrapping the definition

Now that we have defined what a set-theoretic model for our types is, we can choose a particular one that we use to define the rest of the system. Suppose that there exists at least one pair $(D, [\![]\!])$ that satisfies the conditions of set-theoretic model, and choose any of them, no matter the one. Let us call this model the *bootstrap model*. This bootstrap model defines a particular subtyping relation on our set of types $\mathscr{T}$:

$$s \leq t \qquad \Longleftrightarrow \qquad [\![s]\!] \subseteq [\![t]\!]$$

We can then pick any language that uses the types in $\mathscr{T}$ (and whose semantics conforms with the intuition underlying the model condition on function types), define its typing rules and use in the subsumption rule the subtyping relation $\leq$ we have just defined. We write $\Gamma \vdash e : t$ for the typing judgement of the

language. In this paper, we will consider a λ-calculus with overloaded functions and dynamic type-dispatch. See Section 3.1 for the syntax of the calculus, Section 3.3 for its type system and Section 3.2 for its semantics (which depends on the type system because of the dynamic type-dispatch construction).

2.8 Closing the circle

In order to obtain type-safety for our calculus, we want the type system to enjoy properties such as subject reduction (Theorem 8) and progress (Theorem 9) stated in Section 5.1. Because of the subsumption rule in the type system, this can only be obtained if our definition of set-theoretic models is meaningful with respect to the semantics of our calculus. This is a first sanity-check for our notion of model.

But there is another important question: what are the relations between the bootstrap model and the calculus? And in particular, what is the relation between the bootstrap model and the values of the calculus? Have we lost all the intuition underlying the "types as sets of values" interpretation?

To answer these questions, we consider a new interpretation of types as sets of values in the calculus:

$$[\![t]\!]_{\mathscr{V}} = \{v \mid \vdash v : t\}$$

A second sanity-check for our notion of model is then to require that this interpretation $[\![\_]\!]_{\mathscr{V}}$ is a model. If this is the case, we can use it to define a new subtyping relation on $\mathscr{T}$:

$$s \leq_{\mathscr{V}} t \qquad \Longleftrightarrow \qquad [\![s]\!]_{\mathscr{V}} \subseteq [\![t]\!]_{\mathscr{V}}$$

We could imagine to start again the process, that is to use this subtyping relation in the subsumption rule of our language, and use the resulting sets of values to define yet another subtyping relation and so on. But this is not necessary as the process has already converged. This is stated by one of the central results of our work (Theorem 12 in Section 5.2):

$$s \leq t \qquad \Longleftrightarrow \qquad s \leq_{\mathscr{V}} t$$

that is, the subtyping relation induced by the bootstrap model already defines the subtyping relation of the "types as sets of values" model of the resulting calculus. We have closed the circle we broke.

3 The calculus

In this section, we define formally the syntax of types and expression in our calculus (Section 3.1), the semantics (Section 3.2) and the type system (Section 3.3). The semantics actually depends on the type-system, which in turn depends on a subtyping relation to be defined (next section). As a consequence, we consider here the subtyping relation as a parameter of the definitions of the type system and of the semantics.

3.1 Syntax

Expressions To define the calculus, we choose a set of constants $\mathscr{C}$ ranged by the meta-variable c (they will be elements of basic types).

The terms of the calculus are called expressions and are defined by the following grammar.

e	$::=$	c	constant
	$\mid$	(e, e)	pair
	$\mid$	$\mu f(t{\to}t; \ldots; t{\to}t).\lambda x.e$	abstraction
	$\mid$	x	variable
	$\mid$	$e\ e$	application
	$\mid$	$(x = e \in t\ ?\ e \mid e)$	dynamic type dispatch
	$\mid$	$\pi_i(e)$	projection ($i \in \{1, 2\}$)
	$\mid$	$\mathtt{rnd}(t)$	non-deterministic choice

where t ranges over types, defined in the next paragraph.

We write $\mathscr{E}$ for the set of expressions. The syntax for the calculus deserves a few comments. We introduce an explicit construction for recursive functions, which combines λ-abstraction and a fix-point operator. The reason is that we want to express non-terminating expressions, but still restricting recursion only to functions. The identifiers f and x act as binders in the body of the function. The λ-abstraction comes with an non-empty sequence of function types (we call it the *interface* of the function): if more than one type is given, we are in presence of an overloaded function.

The non-deterministic choice construction $\mathtt{rnd}(t)$ picks an arbitrary expression of type t. We introduced this operator in the calculus in order to demonstrate subtle typing issues coming from non-determinism.

Types Types are essentially those introduced in Section 2.6 (modulo boolean equivalence) to which we add basic types (the types of constant expressions). In order to simplify the presentation of recursive types, we are going to consider potentially *infinite regular* terms produced by the following signature:

t	$::=$	b	basic type
	$\mid$	$t{\times}t$	product type
	$\mid$	$t{\to}t$	function type
	$\mid$	$t{\vee}t$	union type
	$\mid$	$\neg t$	complement type
	$\mid$	$\mathbb{0}$	empty type

By regular, we mean that terms have only but a finite number of different sub-terms. The meta-variable b ranges over a fixed set of basic types. We write $t_1 \setminus t_2$ as an abbreviation for $t_1 {\wedge} \neg t_2$, $t_1 {\wedge} t_2$ as an abbrevation for $\neg(\neg t_1 \vee \neg t_2)$, and $\mathbb{1}$ as an abbreviation for $\neg\mathbb{0}$. We will call atom the immediate applications of type constructors: basic types, product types, function types (these are the "atoms" for boolean combinators). Since we want types to denote sets, we need

to impose some constraints to avoid ill-formed types such as a solution to $t = t \vee t$ (which does not carry any information about the set denoted by the type) or to $t = \neg t$ (which cannot represent any set). Namely, we say that a term is a type if it doesn't contain any infinite branch without an atom. Let's call $\mathscr{T}$ the set of types.

The conditions above says that the binary relation $\triangleright \subseteq \mathscr{T}^2$ defined by $t_1 \vee t_2 \triangleright t_i$, $\neg t \triangleright t$ in noetherian. This gives an induction principle on $\mathscr{T}$ that we will use without any further explicit reference to the relation $\triangleright$.

3.2 Semantics

Because of the dynamic type dispatch, the semantics of the calculus depends on its type system. For now, we simply assume that a relation between expressions and types, written $\vdash e : t$ is given. It will be defined in the next section.

Definition 1 *An expression e is a value if it is closed (no free variable), well-typed ($\vdash e : t$ for some type t), and produced by the following grammar:*

$$v ::= c \mid (v, v) \mid \mu f(\ldots).\lambda x.e$$

We write $\mathscr{V}$ for the set of all values.

We define a small-step operational call-by-value semantics $\rightsquigarrow$ for the calculus. There are four basic reduction rules (we write $e[x_1 := e_1; x_2 := e_2; \ldots]$ for the expression obtained from e by a capture-avoiding substitution of x_i by e_i):

$$\begin{array}{l} ev \rightsquigarrow e[f := e'; x := v] \quad \text{if } e = \mu f(\ldots).\lambda x.e' \\ (x = v \in t \ ?\ e_1 | e_2) \rightsquigarrow \begin{cases} e_1[x := v] & \text{if } \vdash v : t \\ e_2[x := v] & \text{if } \vdash v : \neg t \end{cases} \\ \pi_i(v_1, v_2) \rightsquigarrow v_i \\ \texttt{rnd}(t) \rightsquigarrow e \quad \text{if } \vdash e : t \end{array}$$

The relation $\rightsquigarrow$ is further extended by an inductive context rule:

$$C[e] \rightsquigarrow C[e'] \quad \text{if } e \rightsquigarrow e'$$

where the notion of (immediate) context is defined by:

$$\begin{array}{lcl} C[] & ::= & ([], e) \mid (e, []) \\ & \mid & []e \mid e[] \\ & \mid & (x = [] \in t \ ?\ e | e) \mid (x = e \in t \ ?\ [] | e) \mid (x = e \in t \ ?\ e | []) \\ & \mid & \pi_i([]) \\ & \mid & \mu f(\ldots).\lambda x.[] \end{array}$$

As usual, a type safety result will be obtained by a combination of two lemmas: subject reduction (or type preservation) and progress (closed and well-typed expressions which are not values can be reduced).

The reduction rule for application requires the argument to be a value (call-by-value). In order to understand why, let us consider the application

$(\mu f(t \to t\times t; s \to s\times s).\lambda x.(x,x))(\texttt{rnd}(t\vee s))$. The type system will assign to the abstraction the type $(t\to t\times t)\wedge(s\to s\times s)$. A set-theoretic reasoning shows that this type is a subtype of $(t\vee s) \to ((t\times t)\vee(s\times s))$. The type system also assigns to the argument $\texttt{rnd}(t\vee s)$ the type $t\vee s$. It will thus also assign the type $(t\times t)\vee(s\times s)$ to the application. If the semantics permits to reduce this application, we would get as a result the expression $(\texttt{rnd}(t\vee s), \texttt{rnd}(t\vee s))$ whose most precise static type is $(t\vee s)\times(t\vee s)$. Clearly, this type is (in general) a strict supertype of $(t\times t)\vee(s\times s)$. So, if the semantics does not force the argument to be a value in order to reduce an application, we could not obtain the subject reduction lemma.

Similarly, the reduction rule for projection requires its argument to be a value. To understand why, consider the expression $e = \pi_1(e_1, e_2)$ where e_1 is an expression of type e_1 and e_2 is a looping expression of type $\mathbb{0}$ (e.g. $(\mu f(\mathbb{1} \to \mathbb{0}).\lambda x.fx)c$). The type system will assign the type $t_1\times\mathbb{0}$ to e, but in our system $t_1\times\mathbb{0}$ is an empty type because, intuitively, a set-theoretic Cartesian product with an empty component is itself empty. If e could be reduced to e_1, it would be a violation of type preservation.

The same argument applies to the dynamic type dispatch. If we allowed to reduce $(x = e \in t\ ?\ e_1|e_2)$ to $e_1[x := e]$ when $\vdash e : t$, even if e is not a value, we could break type preservation. Consider for instance the case where $\vdash e : \mathbb{0}$. In this case, the type system does not check anything about the branches e_1 and e_2 (the reason for this is explained in details later on) and so e_1 could be ill-typed. Note that when e is a value, then the dynamic type dispatch can always be reduced. Indeed, because our type connectives will be interpreted in a set-theoretic way, we always have $\vdash v : t$ or $\vdash v : \neg t$ (for any value v and any type t).

3.3 Type system

The semantics we just introduced depends on the typing judgment $\Gamma \vdash e : t$ where Γ is a finite mapping from variables to types (we write $\vdash e : t$ when Γ is empty). This judgment, in turn, depends on a subtyping relation $\leq$ between types that we are going to introduce later on. For now, we assume it is a parameter of the type system.

For each constant c, we assume given a basic type b_c. The rules are:

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}\ (subsum) \qquad \frac{}{\Gamma \vdash c : b_c}\ (const) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ (var)$$

$$\frac{\Gamma \vdash e : t_1\times t_2}{\Gamma \vdash \pi_i(e) : t_i}\ (proj) \qquad \frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}\ (appl)$$

$$\frac{\begin{array}{l} t = \bigwedge_{i=1..n} (t_i\to s_i)\wedge \bigwedge_{j=1..m} \neg(t'_j\to s'_j) \not\simeq \mathbb{0} \\ \forall i = 1..n.\Gamma, (f : t), (x : t_i) \vdash e : s_i \end{array}}{\Gamma \vdash \mu f(t_1\to s_1; \ldots; t_n\to s_n).\lambda x.e : t}\ (abstr)$$

$$\frac{\Gamma \vdash e : t_0 \qquad \left\{ \begin{array}{lcl} t_0 \not\leq \neg t & \Rightarrow & \Gamma, (x : t_0 \wedge t) \vdash e_1 : s \\ t_0 \not\leq t & \Rightarrow & \Gamma, (x : t_0 \setminus t) \vdash e_2 : s \end{array} \right.}{\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s} \ (case)$$

The rule (*subsum*) causes the type system to depend on the subtyping relation to be defined. The rules (*const*), (*pair*), (*var*), (*proj*), (*rnd*), and (*appl*) are standard or straightforward.

The rule (*abstr*) is a little bit tricky. Each arrow type $t_i \to s_i$ in the function interface is interpreted as a constraint to be checked. The body of the abstraction is thus type-checked once for each such function. When considering the type $t_i \to s_i$, the variable x is assumed to have type t_i and the body is checked to have type s_i. Also, the variable f is assumed to have type t, which is also the type given to the whole function. Quite intuitively, this type is obtained by taking the intersection of all the types $t_i \to s_i$. But we also add to this intersection any finite number of complement of arrow types, provided the type t does not become empty. This might sound surprising, but the reason is actually simple: we want types to be interpreted as sets of values in such a way that boolean connectives behave as their set-theoretic counterpart. In particular, the union of t and $\neg t$ must always be equivalent to $\mathbb{1}$, that is, we need to have the following property: $\forall v. \forall t. (\vdash v : t)$ or $(\vdash v : \neg t)$. In particular, since a (closed and well-typed) abstraction is value, it must have type $(t \to s)$ or type $\neg(t \to s)$ for any choice of t and s. If $(t \to s)$ is a supertype of the intersection $\bigwedge t_i \to s_i$, the abstraction is known, thanks to the subsumption rule, to have type $(t \to s)$. Otherwise, we need to provide a way to prove it has type $\neg(t \to s)$. This is why we introduce such complements of arrow types in the rule (*abstr*).

The rule (*case*) is easier to read. First, we need to find a type t_0 for the expression whose result will be dynamically type-checked. If this type has a non-empty intersection with t ($t_0 \not\leq \neg t$), then the first branch might be used. In this case, in order for the whole expression to have type s, we need to check that e_1 has also type s, assuming that x has type $t \wedge t_0$. Indeed, at runtime, the variable x will be bound to a value resulting from the evaluation of e_0. Because of subject reduction, this value is necessarily of type t_0. But in order to type-check e_1, we can also assume that the value has type t. If $t_0 \leq \neg t$, then the first branch cannot be used, and we don't need to type-check e_1. Similarly for e_2, replacing t with $\neg t$. The ability to ignore e_1 and/or e_2 when computing the type for $(e \in t \ ? \ e_1 \mid e_2)$ is important to type-check overloaded function. As an example, consider the abstraction $\mu f(b_1 \to b_1; b_2 \to b_2).\lambda x.(x \in b_1 \ ? \ c_1 \mid c_2)$ where b_1 and b_2 are two non-intersecting basic types and c_1 (resp. c_2) is a constant of type b_1 (resp. b_2). The rule (*abstr*), when it considers the arrow type $b_1 \to b_1$, checks that the body has type b_1 assuming that x has type b_1. Clearly, the typing rule for the dynamic type dispatch must discard in this case the type of the second branch.

As an aside note that the use of the *ex falso quodlibet* rule yields a simpler

formulation of the *case* rule:

$$\frac{}{\Gamma, x:\mathbb{0} \vdash e : t}\ (efq) \qquad \frac{\Gamma \vdash e : t_0 \quad \Gamma, (x : t_0 \mathbin{\wedge} t) \vdash e_1 : s \quad \Gamma, (x : t_0 \mathbin{\backslash} t) \vdash e_2 : s}{\Gamma \vdash (x = e \in t\ ?\ e_1 | e_2) : s}\ (case)$$

The reason why we preferred the previous formulation is that it permits a stronger and simpler substitution lemma. A second reason to prefer the previous formulation is that simpler (*case*) rule above does not easily extend to the full version of $\mathbb{C}$Duce with general pattern matching, since it would need special treatment for patterns without any free variable (since these would not produce any $x : \mathbb{0}$ hypothesis in the environment).

4 Subtyping

At this point, we have given the calculus a semantics which depends on its type system, which, in turn, depends on a subtyping relation still to be defined.

The last missing step to complete the definition of our system is the subtyping relation. This will be defined by formalizing the ideas we outlined in Sections 2.6-2.8.

4.1 Set-theoretic interpretations of types

Definition 2 *A* set-theoretic interpretation *of $\mathscr{T}$ is given by a set D and a function $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ such that, for any types t_1, t_2, t:*

- $[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!]$
- $[\![\neg t]\!] = D \backslash [\![t]\!]$
- $[\![\mathbb{0}]\!] = \varnothing$

(A consequence of the conditions is that $[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!]$, $[\![t_1 \mathbin{\backslash} t_2]\!] = [\![t_1]\!] \backslash [\![t_2]\!]$, and $[\![\mathbb{1}]\!] = D$.)

This definition does not say anything about the interpretation of atoms. Actually, using an induction on types, we see that set-theoretic interpretations with domain D correspond univocally to functions from atoms to $\mathscr{P}(D)$.

A set-theoretic interpretation $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ induces a binary relation $\leq_{[\![]\!]} \subseteq \mathscr{T}^2$ defined by:

$$t \leq_{[\![]\!]} s \iff [\![t]\!] \subseteq [\![s]\!]$$

This relation actually only depends on the set of empty types. Indeed, we have: $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \cap (D \backslash [\![t_2]\!]) = \varnothing \iff [\![t_1 \wedge \neg t_2]\!] = \varnothing$. We also get properties of the relation $\leq_{[\![]\!]}$ « for free », such as its transitivity, or the monotonicity of the $\vee$ and $\wedge$ constructors, and so on.

4.2 Models of types

We are going to define a notion of model of the type algebra. Intuitively, a model is a set-theoretic interpretation such that type constructors are interpreted in such as way that the induced relation $\leq_{[\![]\!]}$ capture their essence (in the type system of the calculus), at least as long as subtyping is concerned.

As we explained in Section 2.6, the way to formalize it consists in associating to the interpretation $[\![\_]\!]$ another interpretation $\mathbb{E}(\_)$, called extensional, and then to require, for $[\![\_]\!]$ to be a model, that $[\![\_]\!]$ and $\mathbb{E}(\_)$ behave the same for what concerns subtyping (that is: $[\![t]\!] \subseteq [\![s]\!] \iff \mathbb{E}(t) \subseteq \mathbb{E}(s)$ or, equivalently, $[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$).

For any basic type b, we assume given a set of constants $\mathbb{B}[\![b]\!] \subseteq \mathscr{C}$ whose elements are called constants of type b. Note that for two basic types b_1, b_2, the sets $\mathbb{B}[\![b_i]\!]$ can have a non-empty intersection. For any constant c, we assume that the type b_c is a singleton: $\mathbb{B}[\![b_c]\!] = \{c\}$.

A product type $t_1 \times t_2$ will of course be interpreted extensionally as the Cartesian product $[\![t_1]\!] \times [\![t_2]\!]$.

Things are more complicated for a function type $t_1 \to t_2$. Its extensional interpretation should be the set of set-theoretic functions (that is, functional graphs) f such that $\forall d.\ d \in [\![t_1]\!] \Rightarrow f(d) \in [\![t_2]\!]$. However, the calculus we have in mind can express non-terminating and/or non-deterministic functions as well. This suggests to consider arbitrary binary relations instead of just functional graphs. Also, the calculus has a notion of type error: it is not possible to apply an arbitrary function to an arbitrary value. We are going to take Ω as a special element to denote this type error. Following this discussion, we interpret the function type $t_1 \to t_2$ as the set of binary relations $f \subseteq D \times D_\Omega$ (where $D_\Omega = D + \{\Omega\}$) such that $\forall (d, d') \in f.\ d \in [\![t_1]\!] \Rightarrow d' \in [\![t_2]\!]$.

Definition 3 *If D is a set and X, Y are subsets of D, we write D_Ω for $D + \{\Omega\}$ and define $X \to Y$ as:*

$$X \to Y = \{f \subseteq D \times D_\Omega \mid \forall (d, d') \in f.\ d \in X \Rightarrow d' \in Y\}$$

Note that if we replace D_Ω with D in this definition, then $X \to Y$ is always a subset of $D \to D$. As we will see shortly, this would imply that any arrow type is a subtype of $\mathbb{1} \to \mathbb{1}$. Thanks to the subsumption rule, the application of any well-typed function to any well-typed argument would then be itself well-typed. Clearly, this would break type-safety of the calculus. With Definition 3, instead, we have $X \to Y \subseteq D \to D$ if and only if $D = X$.

We can now give the formal definition of the extensional interpretation associated to a set-theoretic interpretation.

Definition 4 *Let $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ be a set-theoretic interpretation. We define its associated* extensional interpretation *as the unique set-theoretic interpretation $\mathbb{E}(\_) : \mathscr{T} \to \mathscr{P}(\mathbb{E}D)$ (where $\mathbb{E}D = \mathscr{C} + D^2 + \mathscr{P}(D \times D_\Omega)$) such that:*

$$\begin{array}{lcll} \mathbb{E}(b) & = & \mathbb{B}[\![b]\!] & \subseteq \mathscr{C} \\ \mathbb{E}(t_1 \times t_2) & = & [\![t_1]\!] \times [\![t_2]\!] & \subseteq D^2 \\ \mathbb{E}(t_1 \to t_2) & = & [\![t_1]\!] \to [\![t_2]\!] & \subseteq \mathscr{P}(D \times D_\Omega) \end{array}$$

Finally, we can formalize the fact that a set-theoretic interpretation induces the same subtyping relation as if the type constructors were interpreted in an extensional way.

Definition 5 *A set-theoretic interpretation* $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ *is a* model *if it induces the same subtyping relation as its associated extensional interpretation:*

$$\forall t_1, t_2 \in \mathscr{T}.\ [\![t_1]\!] \subseteq [\![t_2]\!] \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$$

Thanks to a remark in Section 4.1, the condition for a set-theoretic interpretation to be a model can be reduced to:

$$\forall t \in \mathscr{T}.\ [\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$$

At this point, we can derive many properties about $\leq$ which directly follow from the fact that it is induced by a model. For instance, the co-/contra-variance of the arrow type constructor, and equivalences such as $(t_1 \to s) \wedge (t_2 \to s) \simeq (t_1 \vee t_2) \to s$, can be immediately derived from the definition of the extensional interpretation. The meta-theoretic study of the system relies in a crucial way on many of such properties. With a more axiomatic approach for defining the subtyping relation, e.g. by a system of inductive or coinductive rules, we would probably need much more work to establish these properties, and we would not have the same level of trust that we did not forget any rule.

4.3 Well-foundedness

The notion of model captures the intended local behavior of type constructors with respect to subtyping. However, it fails to capture a global property of the calculus, namely that values are *finite* binary trees (where leaves are either constants or abstractions). For instance, let us consider the recursive type $t = t \times t$. Intuitively, a value v has this type if and only if it is a pair (v_1, v_2) where v_1 and v_2 also have type t. To build such a value, we would need to consider an infinite tree, which is ruled out. As a consequence, the type t contains no value.

We will introduce a new criterion to capture this property of finite decomposition of pairs.

Definition 6 *A set-theoretic interpretation* $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ *is* structural *if:*

- $D^2 \subseteq D$
- *for any types* t_1,t_2: $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$
- *The binary relation on* D *induced by* $(d_1, d_2) \triangleright d_i$ *is noetherian.*

Definition 7 *A model* $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ *is* well-founded *if it induces the same subtyping relation as a structural set-theoretic interpretation.*

5 Main results

Let us fix an arbitrary model $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$, which we call the bootstrap model. It induces a subtyping relation, which we simply write $\leq$. In turn, this subtyping relation defines a typing judgment $\Gamma \vdash e : t$ for the calculus and thus also a notion of value and a reduction relation $e \rightsquigarrow e'$. We can now state four groups of theoretical results about our system. This first group (Section 5.1) expresses the fact that our notion of models implies that the type system and the semantics are mutually coherent. The second group (Section 5.2) justifies our approach for defining the subtyping relation with a detour through the notion of models: indeed, we can *in fine* re-interpret types as sets of values, and this creates a new model equivalent to the bootstrap model (if it is well-founded). The third group of results (Section 5.3) shows that the notion of model is not void, by expressing the existence of (several different) models satisfying the various conditions. Finally, we focus (Section 5.4) on the effectiveness of the subtyping and typing relations and devise simple subtyping algorithms.

5.1 Type soundness

As announced earlier, we have the two classical lemmas which entail type soundness.

Theorem 8 (Subject reduction) *Let e be an expression and t a type. If $(\Gamma \vdash e : t)$ and $(e \rightsquigarrow e')$, then $(\Gamma \vdash e' : t)$.*

Theorem 9 (Progress) *Let e be a well-typed closed expression. If e is not a value, then there exists an expression e' such that $e \rightsquigarrow e'$.*

It is worth noticing that the proof of Theorem 9 (given in Section 6.6) does not use reductions under abstractions or inside the branches of dynamic type dispatch, thus the result holds true also in that case. Of course, subject reduction holds also if these reductions are disallowed. This means that a weak reduction strategy (as implemented typically in programming languages) enjoys type soundness, too. In the setting of programming languages, proving the subject reduction property also for a semantics that includes strong reduction rules is useful because these rules correspond to possible compile-time optimizations.

Theorem 10 *For every types t and t_1 such that $t \leq t_1 \to \mathbb{1}$, there exists a type t_2 such that, for every value v:*

$$\vdash v : t_2 \iff \exists v_f, v_x.\ (v_f v_x \overset{\star}{\rightsquigarrow} v) \wedge (\vdash v_f : t) \wedge (\vdash v_x : t_1)$$

This type is the smallest solution to the equation $t \leq t_1 \to s$.

The type s in the statement of the theorem above represents exactly all the possible results (i.e. is the set of all values that) we may get when applying a closed expression e_1 of type t_1 to a closed expression e_2 of type t_2. Since $t_1 \leq t_2 \to s$, the type system allows us to derive type s for the application $e_1 e_2$. In other words, the typing rule (*appl*) is *locally exact*: it does not introduce any new approximation to those already made when typing its arguments.

5.2 Closing the loop

The type system naturally defines a new interpretation of types as sets of values:

$$[\![\_]\!]_{\mathscr{V}} : \mathscr{T} \to \mathscr{P}(\mathscr{V}), t \mapsto \{v \mid \vdash v : t\}$$

It turns out that this interpretation satisfies the conditions of Definitions 2 and 6:

Theorem 11 *The function $[\![\_]\!]_{\mathscr{V}}$ is a structural set-theoretic interpretation.*

A natural question is whether this set-theoretic interpretation is a model. If this is the case, we would like to compare the subtyping relation it induces with the one used to define the type system (which was induced by the bootstrap model). The following theorem answers both questions.

Theorem 12 *The following properties are equivalent:*

1. *The interpretation $[\![\_]\!]_{\mathscr{V}}$ is a model.*
2. *The interpretation $[\![\_]\!]_{\mathscr{V}}$ and $[\![\_]\!]$ induce the same subtyping relation.*
3. *The bootstrap model $[\![\_]\!]$ is well-founded.*

When the interpretation $[\![\_]\!]_{\mathscr{V}}$ is a model, we could use it as a new bootstrap model, define a new type system, and so one. The theorem says that it is useless, because the old and the new bootstrap model induce the same subtyping relation.

Note that the type soundness results does not depend on the fact that the interpretation $[\![\_]\!]_{\mathscr{V}}$ is a model. It holds even if the bootstrap model is not well-founded.

5.3 Construction of models

All the results above would be void if we could not build a model. In this section, we actually build models with specific properties. Models can be compared by the amount of subtyping they allow. If $[\![\_]\!]_1$ and $[\![\_]\!]_2$ are two models, we write $[\![\_]\!]_1 \preceq [\![\_]\!]_2$ if:

$$\forall t, s \in \mathscr{T}. [\![t]\!]_1 \leq [\![s]\!]_1 \Rightarrow [\![t]\!]_2 \leq [\![s]\!]_2$$

A model $[\![\_]\!]_2$ is universal if $[\![\_]\!]_1 \preceq [\![\_]\!]_2$ for any other model $[\![\_]\!]_1$. Clearly, two universal models induce the same subtyping relation.

Theorem 13 *There exists a well-founded and universal model.*

The next theorem shows that the notions of universality and well-foundedness are not automatic.

Theorem 14 *There exists a model which is not well-founded. There exists a well-founded model which is not universal.*

5.4 Decidability results

Finally, our system would be of little practical use if we were not able to decide the subtyping and typing relations. Fortunately, the decidability of the inclusion of basic types implies the following theorem.

Theorem 15 *The subtyping relation induced by universal models is decidable.*

The proof of decidability (Section 6.9) essentially relies on three components: (*i*) the regularity of types, (*ii*) some algebraic properties of universal models, and (*iii*) the equivalence between subtyping and type emptiness problems (remember that $s \leq t \iff s\backslash t \simeq \mathbb{0}$.). The algebraic properties of the model can be used to decompose a type t into a set of types t_i's such that: (*i*) $t \simeq \mathbb{0}$ if and only if all $t_i \simeq \mathbb{0}$ and (*ii*) the t_i's are boolean combinations of sub-terms of t (Section 6.2). We also introduce the concept of *simulation* (Section 25) which characterizes sets of types that are closed with respect to the previous decomposition. By construction a type is equivalent to $\mathbb{0}$ if and only if there exists a simulation containing it (the simulation representing a co-inductive proof of its emptiness). A regular type has only a finite number of sub-terms, therefore it suffices to enumerate all the possible sets of boolean combinations of its sub-terms and test whether any of them is a simulation (which is decidable for finite sets).

Decidability of subtyping does not immediately yield decidability of the typing relation, the problem being that the use of the negated arrows in the typing rule (*abstr*) makes the minimum typing property fail. Therefore we need to introduce a new syntactic category, type schemes: a type-scheme represents the set of all the types of a well typed expression (Section 6.12). This technical construction allows us to state the decidability of the type-checking problem.

Theorem 16 *When the subtyping relation is decidable, the type checking problem (deciding whether* $\Gamma \vdash e : t$ *for given* Γ, e, t*) is decidable.*

6 Formal development

In this section, we establish the theorems stated in the previous section and other intermediate lemmas.

6.1 Disjunctive normal forms for types

We write $\mathscr{A}$ for atoms and we use the meta-variable a to range over atoms. There are three kinds of atoms (and values), which we denote by the meta-variable u ranging over the set $U = \{\mathbf{prod}, \mathbf{fun}, \mathbf{basic}\}$.

We write $\mathscr{A}_{\mathbf{fun}}$ for atoms of the form $t_1 \mathbf{\rightarrow} t_2$, $\mathscr{A}_{\mathbf{prod}}$ for atoms of the form $t_1 \mathbf{\times} t_2$, and $\mathscr{A}_{\mathbf{basic}}$ for basic types. We have $\mathscr{A} = \mathscr{A}_{\mathbf{fun}} + \mathscr{A}_{\mathbf{prod}} + \mathscr{A}_{\mathbf{basic}}$. For what concerns values, their kinding too is straightforward: values of the form c, (v_1, v_2), and $\mu f(\ldots).\lambda x.\mathrm{e}$ have respectively kind $\mathbf{basic}, \mathbf{prod}$, and $\mathbf{fun}$.

Every type can be seen as a finite boolean combination of atoms. It is convenient to work with disjunctive normal forms.

Definition 17 *A (disjunctive)* normal formal τ *is a finite set of pairs of finite sets of atoms, that is, an element of* $\mathscr{P}_f(\mathscr{P}_f(\mathscr{A}) \times \mathscr{P}_f(\mathscr{A}))$ *(where* $\mathscr{P}_f$ *denotes the finite powerset).*

If $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ *is an arbitrary set-theoretic interpretation and* τ *a normal form, we define* $[\![\tau]\!]$ *as:*

$$[\![\tau]\!] = \bigcup_{(P,N)\in\tau} \bigcap_{a\in P} [\![a]\!] \cap \bigcap_{a\in N} (D\backslash[\![a]\!])$$

(Note that, with the convention that an intersection over an empty set is taken to be D*,* $[\![\tau]\!] \subseteq D$*.)*

Lemma 18 *For every type* $t \in \mathscr{T}$*, it is possible to compute a normal form* $\mathscr{N}(t)$ *such that for every set-theoretic interpretation* $[\![\_]\!]$*,* $[\![t]\!] = [\![\mathscr{N}(t)]\!]$*.*

Proof: We will actually define two functions $\mathscr{N}$ and $\mathscr{N}'$, both from types to $\mathscr{P}_f(\mathscr{P}_f(\mathscr{A}) \times \mathscr{P}_f(\mathscr{A}))$, by mutual induction over types.

$$\begin{array}{lcl}
\mathscr{N}(\mathbb{0}) & = & \varnothing \\
\mathscr{N}(a) & = & \{(\{a\},\varnothing)\} \\
\mathscr{N}(t_1 \vee t_2) & = & \mathscr{N}(t_1) \cup \mathscr{N}(t_2) \\
\mathscr{N}(\neg t) & = & \mathscr{N}'(t) \\
\mathscr{N}'(\mathbb{0}) & = & \{(\varnothing,\varnothing)\} \\
\mathscr{N}'(a) & = & \{(\varnothing,\{a\})\} \\
\mathscr{N}'(t_1 \vee t_2) & = & \{(P_1 \cup P_2, N_1 \cup N_2) \mid (P_1,N_1) \in \mathscr{N}'(t_1), (P_2,N_2) \in \mathscr{N}'(t_2)\} \\
\mathscr{N}'(\neg t) & = & \mathscr{N}(t)
\end{array}$$

We check by induction over the type t the following property:

$$[\![t]\!] = [\![\mathscr{N}(t)]\!] = D\backslash[\![\mathscr{N}'(t)]\!]$$

□

As an example, consider the type $t = a_1 \wedge (a_2 \vee \neg a_3)$ where a_1, a_2, a_3 are three atoms. Then $\mathscr{N}(t) = \{(\{a_1,a_2\},\varnothing), (\{a_1\},\{a_3\})\}$. This corresponds to the fact that t and $(a_1 \wedge a_2) \vee (a_1 \wedge \neg a_3)$ have the same interpretation for any set-theoretic interpretation of the type algebra.

Note that the converse result is true as well: for any normal form τ, we can find a type t such that $[\![t]\!] = [\![\tau]\!]$ for any set-theoretic interpretation. Normal forms are thus simply a different, but handy, syntax for types. In particular, we can rephrase in Definition 5 the condition for a set-theoretic interpretation to be a model as: for any normal form τ, $[\![\tau]\!] = \varnothing \iff \mathbb{E}(\tau) = \varnothing$.

For these reason henceforth will will often confound the notions of types and normal form, and we will often speak of the *type* τ, taking the latter as a canonical representative of all the types in $\mathscr{N}^{-1}(\tau)$.

6.2 Study of the subtyping relation

Definition 5 is rather intensional. In this section, we establish a more extensional criterion for a set-theoretic interpretation to be a model.

Let $[\![\_]\!]$ be a set-theoretic interpretation. We are interested in comparing the assertions $\mathbb{E}(\tau) = \varnothing$ and $[\![\tau]\!] = \varnothing$, for a normal form τ. Clearly, $\mathbb{E}(\tau) = \varnothing$ is equivalent to:

$$\forall (P, N) \in \tau.\ \bigcap_{a \in P} \mathbb{E}(a) \subseteq \bigcup_{a \in N} \mathbb{E}(a) \tag{3}$$

Let us write $\mathbb{E}^{\mathbf{basic}} D = \mathscr{C}$, $\mathbb{E}^{\mathbf{prod}} D = D^2$, $\mathbb{E}^{\mathbf{fun}} = \mathscr{P}(D \times D_\Omega)$. We have $\mathbb{E}D = \bigcup_{u \in U} \mathbb{E}^u D$ where $U = \{\mathbf{prod}, \mathbf{fun}, \mathbf{basic}\}$. We can thus rewrite (3) as:

$$\forall u \in U. \forall (P, N) \in \tau.\ \bigcap_{a \in P} (\mathbb{E}(a) \cap \mathbb{E}^u D) \subseteq \bigcup_{a \in N} (\mathbb{E}(a) \cap \mathbb{E}^u D) \tag{4}$$

Since $[\![a]\!] \cap \mathbb{E}^u D = \varnothing$ if $a \notin \mathscr{A}_u$ and $[\![a]\!] \cap \mathbb{E}^u D = [\![a]\!]$ if $a \in \mathscr{A}_u$, we can rewrite (4) as:

$$\forall u \in U. \forall (P, N) \in \tau. (P \subseteq \mathscr{A}_u) \Rightarrow \left(\bigcap_{a \in P} \mathbb{E}(a) \subseteq \bigcup_{a \in N \cap \mathscr{A}_u} \mathbb{E}(a) \right) \tag{5}$$

(where the intersection is taken to be $\mathbb{E}^u D$ when $P = \varnothing$.)

To further decompose these predicates, we will rely on two set-theoretic facts, one for product types, one for arrow types. Let us introduce some new notation and then start with product types.

Notation 19 *Let S_1, S_2 denote two sets such that $S_1 \subseteq S_2$. We use $\overline{S_1}^{S_2}$ to denote the complement of S_1 with respect to S_2, that is $S_2 \backslash S_1$.*

Lemma 20 *Let $(X_i)_{i \in P}, (X_i)_{i \in N}$ (resp. $(Y_i)_{i \in P}, (Y_i)_{i \in N}$) be two families of subsets of D_1 (resp. D_2). Then:*

$$\left(\bigcap_{i \in P} X_i \times Y_i \right) \setminus \left(\bigcup_{i \in N} X_i \times Y_i \right) = \bigcup_{N' \subseteq N} \left(\bigcap_{i \in P} X_i \setminus \bigcup_{i \in N'} X_i \right) \times \left(\bigcap_{i \in P} Y_i \setminus \bigcup_{i \in N \setminus N'} Y_i \right)$$

(with the conventions: $\bigcap_{i \in \varnothing} X_i \times Y_i = D_1 \times D_2$; $\bigcap_{i \in \varnothing} X_i = D_1$ and $\bigcap_{i \in \varnothing} Y_i = D_2$)

Proof: First, we notice that:

$$\overline{X_i \times Y_i}^{D_1 \times D_2} = \left(\overline{X_i}^{D_1} \times D_2 \right) \cup \left(D_1 \times \overline{Y_i}^{D_2} \right)$$

From that we get:

$$\bigcap_{i \in N} \overline{X_i \times Y_i}^{D_1 \times D_2} =$$

$$\bigcup_{N' \subseteq N} \left(\bigcap_{i \in N'} \left(\overline{X_i}^{D_1} \times D_2 \right) \cap \bigcap_{i \in N \setminus N'} \left(D_1 \times \overline{Y_i}^{D_2} \right) \right) =$$

$$\bigcup_{N' \subseteq N} \left(\bigcap_{i \in N'} \overline{X_i}^{D_1} \times \bigcap_{i \in N \setminus N'} \overline{Y_i}^{D_2} \right)$$

And finally:

$$\left(\bigcap_{i\in P} X_i \times Y_i\right) \cap \left(\bigcap_{i\in N} \overline{X_i \times Y_i}^{D_1\times D_2}\right) =$$
$$\bigcup_{N'\subseteq N}\left(\left(\bigcap_{i\in P} X_i \cap \bigcap_{i\in N'} \overline{X_i}^{D_1}\right) \times \left(\bigcap_{i\in P} Y_i \cap \bigcap_{i\in N\setminus N'} \overline{Y_i}^{D_2}\right)\right)$$

We get the expected result by applying De Morgan laws. □

We get an immediate corollary.

Lemma 21 *Let P, N be two finite subsets of $\mathscr{A}_{\mathbf{prod}}$. We have:*

$$\bigcap_{a\in P} \mathbb{E}(a) \subseteq \bigcup_{a\in N} \mathbb{E}(a) \iff$$
$$\forall N' \subseteq N.\ \left[\!\!\left[\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1 \right]\!\!\right] = \varnothing \vee \left[\!\!\left[\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2 \right]\!\!\right] = \varnothing$$

(with the convention $\bigcap_{a\in\varnothing} \mathbb{E}(a) = \mathbb{E}^{\mathbf{prod}}D$).

We will now establish a similar result for arrow types. We first decompose the set-theoretic $\to$ operator (Definition 3) into more primitive operators: powerset, complement, Cartesian product.

Lemma 22 *Let $X, Y \subseteq D$. Then:*

$$X \to Y = \mathscr{P}\left(\overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\right)$$

Proof: The result comes from a simple computation:

$$\begin{aligned} X \to Y &= \{f \subseteq D\times D_\Omega \mid \forall (x,y)\in f.\ \neg(x\in X \wedge y \notin Y)\} \\ &= \{f \subseteq D\times D_\Omega \mid f \cap X \times \overline{Y}^{D_\Omega} = \varnothing\} \\ &= \{f \subseteq D\times D_\Omega \mid f \subseteq \overline{X \times \overline{Y}^{D_\Omega}}^{D\times D_\Omega}\} \end{aligned}$$

□

Lemma 23 *Let $(X_i)_{i\in P}$ and $(X_i)_{i\in N}$ be two families of subsets of D. Then:*

$$\bigcap_{i\in P} \mathscr{P}(X_i) \subseteq \bigcup_{i\in N} \mathscr{P}(X_i) \iff \exists i_o \in N.\ \bigcap_{i\in P} X_i \subseteq X_{i_0}$$

Proof: The $\Leftarrow$ implication is trivial. Let us prove the opposite direction. We assume that $\bigcap_{i\in P} \mathscr{P}(X_i) \subseteq \bigcup_{i\in N} \mathscr{P}(X_i)$. The set $\bigcap_{i\in P} X_i$ belongs to all the $\mathscr{P}(X_i)$ for $i \in P$. It is thus in the union of all the $\mathscr{P}(X_i)$ for $i \in N$. We can thus find some $i_0 \in N$ such that $\bigcap_{i\in P} X_i \in \mathscr{P}(X_{i_0})$, which concludes the proof. □

Lemma 24 *Let P and N be two finite subsets of $\mathscr{A}_{\mathbf{fun}}$. Then:*

$$\bigcap_{a\in P} \mathbb{E}(a) \subseteq \bigcup_{a\in N} \mathbb{E}(a)$$
$$\Longleftrightarrow$$
$$\exists (t_0 \rightarrow s_0) \in N.\ \forall P' \subseteq P.\ \left[\!\!\left[t_0 \backslash \left(\bigvee_{t\rightarrow s\in P'} t \right) \right]\!\!\right] = \varnothing \vee \left\{ \begin{array}{l} P \neq P' \\ \left[\!\!\left[\left(\bigwedge_{t\rightarrow s\in P\setminus P'} s \right) \backslash s_0 \right]\!\!\right] = \varnothing \end{array} \right.$$

(with the convention $\bigcap_{a\in\varnothing} \mathbb{E}(a) = \mathbb{E}^{\mathbf{fun}} D$).

Proof: The result follows from Lemmas 22, 23, and 20, by noticing that in the condition $\bigcap_{t\rightarrow s\in P\setminus P'} [\![s]\!] \subseteq [\![s_0]\!]$ which appears, the convention is to interpret the intersection as being D_Ω if $P = P'$, which makes the inclusion impossible. □

Lemmas 21 and 24, together with the property (5) suggest the following definition and give immediatly the result of Theorem 26 below.

Definition 25 (Simulation) *Let $\mathscr{S}$ be an arbitrary set of normal forms. We define another set of normal forms $\mathbb{E}\mathscr{S}$ by:*

$$\mathbb{E}\mathscr{S} = \{\tau \mid \forall u \in U. \forall (P,N) \in \tau.\ (P \subseteq \mathscr{A}_u \Rightarrow C_u^{P, N\cap \mathscr{A}_u})\}$$

where:

$$C^{P,N}_{\mathbf{basic}} \quad ::= \quad \mathscr{C} \cap \bigcap_{b\in P} \mathbb{B}[\![b]\!] \subseteq \bigcup_{b\in N} \mathbb{B}[\![b]\!]$$

$$C^{P,N}_{\mathbf{prod}} \quad ::= \quad \forall N' \subseteq N. \begin{cases} \mathscr{N}\left(\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1\right) \in \mathscr{S} \\ \qquad\qquad \vee \\ \mathscr{N}\left(\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2\right) \in \mathscr{S} \end{cases}$$

$$C^{P,N}_{\mathbf{fun}} \quad ::= \quad \exists t_0\to s_0 \in N.\ \forall P' \subseteq P. \begin{cases} \mathscr{N}\left(t_0 \wedge \bigwedge_{t\to s\in P'} \neg t\right) \in \mathscr{S} \\ \qquad\qquad \vee \\ \begin{cases} P \neq P' \\ \mathscr{N}\left((\neg s_0) \wedge \bigwedge_{t\to s\in P\setminus P'} s\right) \in \mathscr{S} \end{cases} \end{cases}$$

We say that $\mathscr{S}$ is a simulation *if:*

$$\mathscr{S} \subseteq \mathbb{E}\mathscr{S}$$

The intuition is that if we consider the statements of Lemmas 21 and 24 as if they were rewriting rules (from right to left), then $\mathbb{E}\mathscr{S}$ contains all the types that we can deduce in one step reduction to be empty when we suppose that the types in $\mathscr{S}$ are empty. A simulation is thus a set that is already saturated w.r.t. such a rewriting. In particular, if we consider the statements of Lemmas 21 and 24 as inference rules for determining when a type is equal to $\mathbb{0}$, then $\mathbb{E}\mathscr{S}$ is the set of immediate consequences of $\mathscr{S}$, and a simulation is a *self-justifying* set, that is a co-inductive proof of the fact that all its elements are equal to $\mathbb{0}$. Of course this latter property will play a crucial role to decide the subtyping relation (see Section 6.9).

Theorem 26 *Let $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ be a set-theoretic interpretation. We define a set of normal forms $\mathscr{S}$ by:*

$$\mathscr{S} = \{\tau \mid [\![\tau]\!] = \varnothing\}$$

Then:

$$\mathbb{E}\mathscr{S} = \{\tau \mid \mathbb{E}(\tau) = \varnothing\}$$

Corollary 27 *Let $[\![\_]\!]$ be a set-theoretic intepretation of types and $\mathscr{S} = \{\tau \mid [\![\tau]\!] = \varnothing\}$. Then $[\![\_]\!]$ is a model if and only if $\mathscr{S} = \mathbb{E}\mathscr{S}$.*

This Corollary implies that the condition for a set-theoretic interpretation to be a model depends only on the subtyping relation it induces.

Corollary 28 *Let $[\![\_]\!]_1 : \mathscr{T} \to \mathscr{P}(D_1)$ be a model and $[\![\_]\!]_2 : \mathscr{T} \to \mathscr{P}(D_2)$ be a set-theoretic interpretation. Then the following assertions are equivalent:*

- *$[\![\_]\!]_2$ is a model and it induces the same subtyping relation as $[\![\_]\!]_1$.*
- *for any type t, $[\![t]\!]_1 = \varnothing \iff [\![t]\!]_2 = \varnothing$.*

The following lemma, which is an immediate corollary of Lemma 24 gives several properties about subtyping between arrow types in a model, which will be needed for to study the meta-theory of the type system.

Lemma 29 (Strong disjunction for arrows) *Let $\leq$ be the subtyping relation induced by a model, and P,N two finite sets of arrow types. Then:*

$$\bigwedge_{a\in P} a \leq \bigvee_{a\in N} a \iff \exists a_0 \in N. \bigwedge_{a\in P} a \leq a_0$$

If P,N are finite sets of arrow types and if a_0 is an arrow type, then:

$$\begin{cases} \bigwedge_{a\in P} a \not\leq \bigvee_{a\in N} a \\ \bigwedge_{a\in P} a \leq \bigvee_{a\in N\cup\{a_0\}} a \end{cases} \Longrightarrow \bigwedge_{a\in P} a \leq a_0$$

If P,N_1,N_2 are finite sets of arrow types, then:

$$\begin{cases} \bigwedge_{a\in P} a \not\leq \bigvee_{a\in N_1} a \\ \bigwedge_{a\in P} a \not\leq \bigvee_{a\in N_2} a \end{cases} \iff \bigwedge_{a\in P} a \not\leq \bigvee_{a\in N_1\cup N_2} a$$

6.3 Syntactical meta-theory of the type system

In this section and in the following one, we fix a bootstrap model $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$, we write $\leq$ for the induced subtyping relation and $\simeq$ for the associated equivalence relation, and we study the resulting typing judgment $\Gamma \vdash e : t$.

Lemma 30 (Strengthening) *Let Γ_1 and Γ_2 be two typing environments such that for any x in the domain of Γ_1, we have $\Gamma_2(x) \leq \Gamma_1(x)$. If $\Gamma_1 \vdash e : t$, then $\Gamma_2 \vdash e : t$.*

Proof: Induction on the derivation of $\Gamma_1 \vdash e : t$. We simply introduce an instance of the subsumption rule below each instance of the (*var*) rule. □

Lemma 31 (Admissiblity of the intersection rule) *If $\Gamma \vdash e : t_1$ and $\Gamma \vdash e : t_2$, then $\Gamma \vdash e : t_1 \wedge t_2$.*

Proof: By induction on the structure of the two typing derivations.
Let us first consider the case when the last rule applied to one of the two derivations is (*subsum*), say:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash e : s_1} \quad s_1 \leq t_1}{\Gamma \vdash e : t_1} \quad \dfrac{\ldots}{\Gamma \vdash e : t_2}$$

The induction hypothesis gives $\Gamma \vdash e : s_1 \wedge t_2$. But $s_1 \wedge t_2 \leq t_1 \wedge t_2$ because $s_1 \leq t_1$, and a new application of (*subsum*) gives $\Gamma \vdash e : t_1 \wedge t_2$ as expected.
In all the remaining cases, the two derivations ends with an instance of the same rule (which depends on the toplevel constructor of e).
Rules (*const*), (*var*), (*rnd*): Those rules give only one possible type t for e, and $t \wedge t \simeq t$.
Rule (*appl*): The situation is as follows:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash e_1 : t_1 \to t_2} \quad \dfrac{\ldots}{\Gamma \vdash e_2 : t_1}}{\Gamma \vdash e_1 e_2 : t_2} \qquad \dfrac{\dfrac{\ldots}{\Gamma \vdash e_1 : t'_1 \to t'_2} \quad \dfrac{\ldots}{\Gamma \vdash e_2 : t'_1}}{\Gamma \vdash e_1 e_2 : t'_2}$$

The induction hypothesis gives $\Gamma \vdash e_1 : (t_1 \to t_2) \wedge (t'_1 \to t'_2)$ and $\Gamma \vdash e_2 : t_1 \to t'_1$. To conclude, it is enough to check that $(t_1 \to t_2) \wedge (t'_1 \to t'_2) \leq (t_1 \wedge t'_1) \to (t_2 \wedge t'_2)$, which can be proved as follows:

$$\begin{array}{ll} & \mathbb{E}((t_1 \to t_2) \wedge (t'_1 \to t'_2)) \\ = & (\llbracket t_1 \rrbracket \to \llbracket t_2 \rrbracket) \cap (\llbracket t'_1 \rrbracket \to \llbracket t'_2 \rrbracket) \\ = & \{f \in \mathbb{E}^{\mathbf{fun}} D \mid \forall (x,y) \in f. (x \in \llbracket t_1 \rrbracket \Rightarrow y \in \llbracket t_2 \rrbracket) \wedge (x \in \llbracket t'_1 \rrbracket \Rightarrow y \in \llbracket t'_2 \rrbracket)\} \\ \subseteq & \{f \in \mathbb{E}^{\mathbf{fun}} D \mid \forall (x,y) \in f. (x \in \llbracket t_1 \rrbracket \cap \llbracket t'_1 \rrbracket \Rightarrow y \in (\llbracket t_2 \rrbracket \cap \llbracket t'_2 \rrbracket)\} \\ = & \mathbb{E}((t_1 \wedge t'_1) \to (t_2 \wedge t'_2)) \end{array}$$

Rule (*pair*): The situation is as follows:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash e_1 : t_1} \quad \dfrac{\ldots}{\Gamma \vdash e_2 : t_2}}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2} \qquad \dfrac{\dfrac{\ldots}{\Gamma \vdash e_1 : t'_1} \quad \dfrac{\ldots}{\Gamma \vdash e_2 : t'_2}}{\Gamma \vdash (e_1, e_2) : t'_1 \times t'_2}$$

Let $t''_1 = t_1 \wedge t'_1$ and $t''_2 = t_2 \wedge t'_2$. By applying the induction hypothesis twice, we get $\Gamma \vdash e_1 : t''_1$ et $\Gamma \vdash e_2 : t''_2$. The rule (*pair*) gives $\Gamma \vdash (e_1, e_2) : t''_1 \times t''_2$. To conclude, it is enough to see that $t''_1 \times t''_2 \simeq (t_1 \times t_2) \wedge (t'_1 \times t'_2)$. Indeed:

$$\mathbb{E}(t''_1 \times t''_2) = (\llbracket t_1 \rrbracket \cap \llbracket t'_1 \rrbracket) \times (\llbracket t_2 \rrbracket \cap \llbracket t'_2 \rrbracket) = \llbracket t_1 \wedge t_2 \rrbracket \cap \llbracket t'_1 \wedge t'_2 \rrbracket = \mathbb{E}((t_1 \times t_2) \wedge (t'_1 \times t'_2))$$

Rule (*case*): Let us consider this situation:

$$\dfrac{\dfrac{\ldots}{\Gamma \vdash e : t_0} \quad \dfrac{\ldots}{(x : t_i), \Gamma \vdash e_i : s}}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s} \qquad \dfrac{\dfrac{\ldots}{\Gamma \vdash e : t'_0} \quad \dfrac{\ldots}{(x : t'_i), \Gamma \vdash e_i : s'}}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s'}$$

with $t_1 = t_0 \wedge t$, $t_2 = t_0 \setminus t$, $t'_1 = t'_0 \wedge t$, $t'_2 = t'_0 \setminus t$. The induction hypothesis gives: $\Gamma \vdash e : t''_0$ with $t''_0 = t_0 \wedge t'_0$. Let us define $t''_1 = t''_0 \wedge t$ and $t''_2 = t''_0 \setminus t$. Let

$i \in \{1,2\}$. We have $t_i'' \leq t_i$ and thus, according to Lemma 30, $(x : t_i''), \Gamma \vdash e_i : s$. Similarly, we get $(x : t_i''), \Gamma \vdash e_i : s'$, and thus, applying again the induction hypothesis $(x : t_i''), \Gamma \vdash e_i : s''$ where $s'' = s \wedge s'$. Then, with the (*case*) rule, we establish $\Gamma \vdash (x = e \in t \ ? \ e_1 | e_2) : s''$ as expected.
The special cases (where $t_i \simeq \mathbb{0}$ or $t_i' \simeq \mathbb{0}$) are similar.
Rule (*abstr*): Let us consider two applications of the rule (*abstr*) to the same abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$ with the following types:

$$\begin{aligned} t &= \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t_j' \to s_j') \\ t' &= \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=m+1..m'} \neg(t_j' \to s_j') \end{aligned}$$

where $t \not\simeq \mathbb{0}$ and $t' \not\simeq \mathbb{0}$. We define:

$$t'' = \bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m'} \neg(t_j' \to s_j')$$

We have $t'' \simeq t \wedge t'$. We only need to verify that some instance of the rule (*abstr*) allows us to deduce the type t'' for the abstraction. For $i = 1..n$, we have, by hypothesis $(f : t), (x : t_i), \Gamma \vdash e : s_i$, and thus, according to Lemma 30, $(f : t''), (x : t_i), \Gamma \vdash e : s_i$. Then, we check that $t'' \not\simeq \mathbb{0}$, which results immediatly from Lemma 29. In this case, we have not used the induction hypothesis. □

Corollary 32 *Let Γ be a typing environment and e an expression which is well-typed under Γ. Then the set $\{t \in \mathscr{T} \mid (\Gamma \vdash e : t) \vee (\Gamma \vdash e : \neg t)\}$ contains $\mathbb{0}$ and is stable under $\vee$ and $\neg$ (and thus $\wedge$).*

Proof: Let E be the set introduced in the statement. It is clearly stable under $\neg$ and invariant under the equivalence $\simeq$. We have $\Gamma \vdash e : \mathbb{1} = \neg\mathbb{0}$ because of the subsumption rule, and thus $\mathbb{0} \in E$. What remains is to prove that E is stable under $\vee$. So let us take two elements t_1 and t_2 in E. If $\Gamma \not\vdash e : t_1 \vee t_2$, then because of (*subsum*), we get $\Gamma \not\vdash e : t_1$ and $\Gamma \not\vdash e : t_2$. Because t_1 and t_2 are in E, we thus have $\Gamma \vdash e : \neg t_1$ and $\Gamma \vdash e : \neg t_2$. Lemma 31 then gives $\Gamma \vdash e : \neg t_1 \wedge \neg t_2$. And $\neg t_1 \wedge \neg t_2 \simeq \neg(t_1 \vee t_2)$. We have thus proved that $\Gamma \vdash e : t_1 \vee t_2$ or $\Gamma \vdash e : \neg(t_1 \vee t_2)$. □

Lemma 33 (Substitution) *Let $e, e_1, \ldots, e_n$ be expressions, $x_1, \ldots, x_n$ distinct variables, $t, t_1, \ldots, t_n$ types, and Γ a typing environment. Then:*

$$\left\{ \begin{array}{l} (x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t \\ \forall i = 1..n.\ \Gamma \vdash e_i : t_i \end{array} \right. \Rightarrow \Gamma \vdash e[x_1 := e_1; \ldots; x_n := e_n] : t$$

Proof: By induction on the typing derivation for $(x_1 : t_1), \ldots, (x_n : t_n), \Gamma \vdash e : t$. We simply "plug" a copy of the derivation for $\Gamma \vdash e_i : t_i$ wherever the rule (var) is used for variable x_i. □

6.4 Interpreting types as sets of values

The syntactical properties obtained in the previous section are used here to prove some properties about the interpretation of types as sets of values, as defined in Section 5.2: $[\![t]\!]_{\mathscr{V}} = \{v \mid \vdash v : t\}$

Lemma 34 *If $t \leq s$, then $[\![t]\!]_{\mathscr{V}} \subseteq [\![s]\!]_{\mathscr{V}}$. In particular, if $t \simeq s$, then $[\![t]\!]_{\mathscr{V}} = [\![s]\!]_{\mathscr{V}}$.*

Proof: Consequence of the subsumption rule. □

Lemma 35 $[\![\mathbb{0}]\!]_{\mathscr{V}} = \varnothing$.

Proof: We prove that $(\vdash v : t) \Rightarrow t \neq \mathbb{0}$ by induction on the typing derivation. There are four cases to consider (one per value constructor, one for the subsumption rule). All of them are trivial. □

Lemma 36 $[\![t_1 \wedge t_2]\!]_{\mathscr{V}} = [\![t_1]\!]_{\mathscr{V}} \cap [\![t_2]\!]_{\mathscr{V}}$.

Proof: Lemma 34 gives $[\![t_1 \wedge t_2]\!]_{\mathscr{V}} \subseteq [\![t_i]\!]_{\mathscr{V}}$ for $i \in \{1, 2\}$, and thus $[\![t_1 \wedge t_2]\!]_{\mathscr{V}} \subseteq [\![t_1]\!]_{\mathscr{V}} \cap [\![t_2]\!]_{\mathscr{V}}$. Lemma 31 gives the opposite inclusion. □

Lemma 37 (Inversion)

$$\begin{array}{lcl} [\![t_1 \times t_2]\!]_{\mathscr{V}} & = & \{(v_1, v_2) \mid \vdash v_1 : t_1, \vdash v_2 : t_2\} \\ [\![b]\!]_{\mathscr{V}} & = & \{c \mid b_c \leq b\} \\ [\![t \to s]\!]_{\mathscr{V}} & = & \{(\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e) \in \mathscr{V}. \mid \bigwedge_{i=1..n} t_i \to s_i \leq t \to s\} \end{array}$$

Moreover, if v is a value and a is an atom of a different kind, then $\vdash v : \neg a$.

Proof: For the three equalities, the $\supseteq$ inclusion is straightforward.
To prove the three opposite inclusion, let us start with a general remark. A derivation for $\vdash v : t$ can always be described as an instance of the rule corresponding to the kind of v (rule $(const)$ for constants, $(pair)$ for pairs, and $(abstr)$ for abstractions), followed by zero or more instance of $(subsum)$.

That is, we can always find another type $t' \leq t$ such that $\vdash v : t$ is obtained by a direct application of the typing rule corresponding to v. If t is an atom a, then v is necessarily of the same kind as a. Indeed, if v is a pair, then t' is a product type; if v is a constant, t' is a basic type; if v is an abstraction, t' is an intersection of one or more arrow types (and maybe of zero or more negation of arrow types). In all cases, $t' \cap a \simeq \mathbb{0}$ if a and v does not have the same kind, but since $t' \leq a$, this means that $t' \simeq \mathbb{0}$, which is impossible. We also have proved the final remark in the statement of the Lemma (because if a and v does not have the same kind, then $t' \leq \neg a$, and thus $\vdash v : \neg a$).
Case $\vdash v : t_1 \times t_2$:. The value is necessarily a pair (v_1, v_2) such that $\vdash v_1 : t'_1$, $\vdash v_2 : t'_2$, and $t'_1 \times t'_2 \leq t_1 \times t_2$. But $t'_1 \not\simeq \mathbb{0}$ and $t'_2 \not\simeq \mathbb{0}$ because of Lemma 35, and thus $t'_1 \leq t_1$ and $t'_2 \leq t_2$. By subsumption, we get $\vdash v_1 : t_1$ and $\vdash v_2 : t_2$.
Case $\vdash v : b$: The value is necessarily a constant c such that $b_c \leq b$.
Case $\vdash v : t \to s$: The value is necessarily an abstraction $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$. Here, the type t' has the form:

$$\bigwedge_{i=1..n} (t_i \to s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \to s'_j)$$

with $t' \not\simeq \mathbb{0}$ and $t' \leq t \to s$. Lemma 29 thus gives:

$$\bigwedge_{i=1..n} (t_i \to s_i) \leq t \to s$$

□

Lemma 38 $[\![\neg t]\!]_{\mathscr{V}} = \mathscr{V} \setminus [\![t]\!]_{\mathscr{V}}$.

Proof:
We have $(t \wedge \neg t) \simeq \mathbb{0}$ and, thus, $[\![t]\!]_{\mathscr{V}} \cap [\![\neg t]\!]_{\mathscr{V}} = [\![t \wedge \neg t]\!]_{\mathscr{V}} = [\![\mathbb{0}]\!]_{\mathscr{V}} = \varnothing$. So it remains to prove that $[\![t]\!]_{\mathscr{V}} \cup [\![\neg t]\!]_{\mathscr{V}} = \mathscr{V}$, that is:

$$\forall v. \forall t.\ (\vdash v : t) \vee (\vdash v : \neg t)$$

We proceed by induction over the pair (v, t). Thanks to Lemma 32, we can assume that t is an atom a. Lemma 37 gives $\vdash v : \neg a$ if a and v do not have the same kind. Now, we assume they have the same kind.
Case $v = c$: We have $\vdash c : b_c$. The set $\mathbb{E}(b_c)$ is a singleton (namely $\{c\}$), and thus $\mathbb{E}(b_c) \subseteq \mathbb{E}(a)$ or $\mathbb{E}(b_c) \subseteq \mathbb{E}(\neg a)$, that is: $b_c \leq a$ or $b_c \leq \neg a$. By subsumption, we get $\vdash b_c : a$ or $\vdash b_c : \neg a$.
Case $v = (v_1, v_2)$, $a = t_1 \times t_2$: If $\vdash v_1 : t_1$ and $\vdash v_2 : t_2$, we get $\vdash v : a$. Otherwise, say $\not\vdash v_1 : t_1$, we get $\vdash v_1 : \neg t_1$ by the induction hypothesis, and $\vdash v_2 : \mathbb{1}$ always holds, and thus we get $\vdash v : (\neg t_1) \times \mathbb{1}$. We conclude this case by the observation that $(\neg t_1) \times \mathbb{1} \leq \neg(t_1 \times t_2)$.
Case $v = \mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e$, $a = t \to s$: It is easy to see that $\vdash v : a$ if $\bigwedge_{i=1..n} t_i \to s_i \leq a$ and $\vdash v : \neg a$ otherwise. □

Lemma 39 $[\![t_1 \vee t_2]\!]_{\mathscr{V}} = [\![t_1]\!]_{\mathscr{V}} \cup [\![t_2]\!]_{\mathscr{V}}$.

Proof: Using Lemmas 38, 36 and 34, we get: $[\![t_1 \vee t_2]\!]_{\mathscr{V}} = [\![\neg((\neg t_1) \wedge (\neg t_2))]\!]_{\mathscr{V}} = \mathscr{V} \setminus ([\![\neg t_1]\!]_{\mathscr{V}} \cap [\![\neg t_2]\!]_{\mathscr{V}}) = \mathscr{V} \setminus (\mathscr{V} \setminus [\![t_1]\!]_{\mathscr{V}} \setminus [\![t_2]\!]_{\mathscr{V}}) = [\![t_1]\!]_{\mathscr{V}} \cup [\![t_2]\!]_{\mathscr{V}}$. □

From Lemmas 38, 39 and 35, we get that $[\![\_]\!]_{\mathscr{V}}$ is a set-theoretic interpretation.

To conclude the proof of Theorem 11, we need to check that it is structural. Clearly $\mathscr{V}^2 \subseteq \mathscr{V}$ and Lemma 37 gives $[\![t_1 \times t_2]\!]_{\mathscr{V}} = [\![t_1]\!]_{\mathscr{V}} \times [\![t_2]\!]_{\mathscr{V}}$. Also, the relation induced by $(v_1, v_2) \triangleright v_i$ is clearly noetherian.

6.5 Closing the loop

Lemma 40 *For every non-empty and finite family of arrow types $t_1 \to s_1, \ldots, t_n \to s_n$, the expression $\mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.fx$ is a value.*

Proof: Direct application of the typing rules. □

Lemma 41 *In every model, $[\![t]\!] = \varnothing \iff [\![\mathbb{1} \to t]\!] \subseteq [\![\mathbb{1} \to \mathbb{0}]\!]$ holds true.*

Lemma 42 *The set-theoretic interpretation $[\![\_]\!]_{\mathscr{V}}$ is a model if and only if it induces the same subtyping relation as $[\![\_]\!]$.*

Proof: The $\Leftarrow$ implication is given by Corollary 28. Let us assume that $[\![\_]\!]_{\mathscr{V}}$ is a model and prove that $[\![t]\!]_{\mathscr{V}} = \varnothing \iff t \simeq \mathbb{0}$ for any type t. The $\Leftarrow$ implication is given by Lemma 35. Let t be a type such that $[\![t]\!]_{\mathscr{V}} = \varnothing$. Because $[\![\_]\!]_{\mathscr{V}}$ is a model, Lemma 41 gives: $[\![\mathbb{1} \to t]\!]_{\mathscr{V}} \subseteq [\![\mathbb{1} \to \mathbb{0}]\!]_{\mathscr{V}}$. Now we consider the expression $v = \mu f(\mathbb{1} \to t).\lambda x.fx$. According to Lemma 40, it is a value. According to Lemma 37, it is an element of $[\![\mathbb{1} \to t]\!]_{\mathscr{V}}$, and thus also of $[\![\mathbb{1} \to \mathbb{0}]\!]_{\mathscr{V}}$, which means that $\mathbb{1} \to t \leq \mathbb{1} \to \mathbb{0}$ (again Lemma 37), and finally that $t \simeq \mathbb{0}$ (Lemma 41 for the model $[\![\_]\!]$). □

Lemma 43 *If the bootstrap model is well-founded, then $[\![\_]\!]_{\mathscr{V}}$ is a model.*

Proof: Since the type system, and thus $[\![\_]\!]_{\mathscr{V}}$, depends only on the subtyping relation induced by the bootstrap model, we can assume that it is not only well-founded, but also structural. We will use the noetherian relation $\triangleright$ from Definition 6.

We need to prove that, for every type t, $[\![t]\!]_{\mathscr{V}} = \varnothing \iff t \simeq \mathbb{0}$. The $\Leftarrow$ implication is given by Lemma 35. We actually prove by induction (using the $\triangleright$ relation) that for all $d \in D$, the following property holds: $(\forall t \in \mathscr{T}.\ d \in [\![t]\!] \Rightarrow [\![t]\!]_{\mathscr{V}} \neq \varnothing)$.
Consider a type t such that $d \in [\![t]\!]$. If $d = (d_1, d_2) \in D^2$, then it is in the set

$$[\![t]\!] \cap D^2 = \bigcup_{(P,N)\in\mathscr{N}(t)} \left(D^2 \cap \bigcap_{a\in P} [\![a]\!] \setminus \bigcup_{a\in N} [\![a]\!] \right)$$

We can thus find $(P, N) \in \mathscr{N}(t)$ such that $d \in D^2 \cap \bigcap_{a\in P} [\![a]\!] \setminus \bigcup_{a\in N} [\![a]\!]$. Note that if a is an atom which is not a product type, then $D^2 \cap [\![a]\!] = [\![\mathbb{1}\times\mathbb{1}]\!] \cap [\![a]\!] = \varnothing$, because $\mathbb{E}(\mathbb{1}\times\mathbb{1}) \cap \mathbb{E}(a) = \varnothing$. We can thus assume that $P \subseteq \mathscr{A}_{\mathbf{prod}}$, and we have $d \in \bigcap_{t_1\times t_2\in P}([\![t_1]\!] \times [\![t_2]\!]) \setminus \bigcup_{t_1\times t_2\in N}([\![t_1]\!] \times [\![t_2]\!])$. If we write $d = (d_1, d_2)$, then Lemma 20 gives some $N' \subseteq N$ such that $d_1 \in [\![s_1]\!]$ and $d_2 \in [\![s_2]\!]$ for:

$$\begin{cases} s_1 &= \bigwedge_{t_1\times t_2\in P} t_1 \setminus \bigvee_{t_1\times t_2\in N'} t_1 \\ s_2 &= \bigwedge_{t_1\times t_2\in P} t_2 \setminus \bigvee_{t_1\times t_2\in N\setminus N'} t_2 \end{cases}$$

The induction hypothesis applied to d_1 and d_2 gives $[\![s_1]\!]_{\mathscr{V}} \neq \varnothing$ and $[\![s_2]\!]_{\mathscr{V}} \neq \varnothing$, and thus $[\![s_1\times s_2]\!]_{\mathscr{V}} \neq \varnothing$. To conclude this case, we observe that $s_1\times s_2 \leq t$, using again Lemma 20.
Now, we assume that $d \notin D^2 = [\![\mathbb{1}\times\mathbb{1}]\!]$. We thus have $d \in [\![t\setminus\mathbb{1}\times\mathbb{1}]\!]$, which implies that $t\setminus\mathbb{1}\times\mathbb{1} \not\simeq \mathbb{0}$. As a consequence $\mathbb{E}(t\setminus\mathbb{1}\times\mathbb{1}) \neq \varnothing$, and thus $\mathbb{E}(t) \cap (\mathbb{E}D\setminus\mathbb{E}^{\mathbf{prod}}D) \neq \varnothing$. We are in at least one of the two cases:
$\mathbb{E}(t) \cap \mathscr{C} \neq \varnothing$: let $c \in \mathbb{E}(t) \cap \mathscr{C}$. We have $\mathbb{E}(b_c) = \{c\} \subseteq \mathbb{E}(t)$, and thus $b_c \leq t$. We conclude that $\vdash c : t$.
$\mathbb{E}(t) \cap \mathbb{E}^{\mathbf{fun}}D \neq \varnothing$: we have:

$$\mathbb{E}(t) \cap \mathbb{E}^{\mathbf{fun}}D = \bigcup_{(P,N)\in\mathscr{N}(t)\ |\ P\subseteq\mathscr{A}_{\mathbf{fun}}} \left(\mathbb{E}^{\mathbf{fun}}D \cap \bigcap_{a\in P} \mathbb{E}(a) \setminus \bigcup_{a\in N} \mathbb{E}(a) \right)$$

This set is not empty. We can thus find an element (P, N) in $\mathscr{N}(t)$ such that $P = \{t_1\to s_1, \ldots, t_n\to s_n\}$, $N \cap \mathscr{A}_{\mathbf{fun}} = \{t'_1\to s'_1, \ldots, t'_m\to s'_m\}$, and $t' = \bigwedge_{i=1..n} t_i\to s_i \setminus \bigvee_{j=1..m} t'_j\to s'_j \not\simeq \mathbb{0}$. We have $t' \leq t$ and the value $v = \mu f(t_1\to s_1; \ldots; t_n\to s_n).\lambda x.fx$ has type t'. By subsumption, we get $\vdash v : t$. □

Lemmas 43 and 42 entail Theorem 12.

6.6 Type soundness

Here is the proof of the subject reduction property, Theorem 8 in Section 5.

Proof: If $(\Gamma \vdash e : t)$, then we prove by induction on the derivation for $\Gamma \vdash e : t$ that $\forall e'.(e \rightsquigarrow e') \Rightarrow (\Gamma \vdash e' : t)$. We consider the last rule used in the derivation of $\Gamma \vdash e : t$.

Rule (*subsum*): we have $\Gamma \vdash e : s \leq t$ and $e \rightsquigarrow e'$. The induction hypothesis gives $\Gamma \vdash e' : s$, and by subsumption we get $\Gamma \vdash e' : t$.

Rules (*const*),(*var*): the expression e is a constant or a variable. It cannot be reduced.

Rule (*proj*): we have $e = \pi_i(e_0)$, $t = t_i$, $\Gamma \vdash e_0 : t_1 \times t_2$. If e' is obtained by reducing e_0, that is, $e_0 \rightsquigarrow e'_0$ and $e' = \pi_i(e'_0)$, we get, by the induction hypothesis: $\Gamma \vdash e'_0 : t_1 \times t_2$ and thus $\Gamma \vdash e' : t_i$. If e' is obtained by reducing the toplevel π_i in e, then necessarily e_0 is a value (v_1, v_2) (and thus, by Lemma 37: $\Gamma \vdash v_i : t_i$), and $e' = v_i$. We get $\Gamma \vdash e' : t_i$.

Rule (*rnd*): we have $e = \mathtt{rnd}(t)$. The reduction rule for this expression gives $\vdash e' : t$, which implies $\Gamma \vdash e' : t$ by Lemma 30.

Rule (*pair*): we have $e = (e_1, e_2)$, $t = t_1 \times t_2$, and $\Gamma \vdash e_i : t_i$ for $i = 1..2$. The only possible way to reduce e is to reduce one of the e_i, say $e' = (e'_1, e_2)$ where $e_1 \rightsquigarrow e'_1$. The induction hypothesis gives $\Gamma \vdash e'_1 : t_1$, and we get $\Gamma \vdash e' : t_1 \times t_2$.

Rule (*appl*): we have $e = e_1 e_2$, $\Gamma \vdash e_1 : s \to t$ and $\Gamma \vdash e_2 : s$. If e' is obtained by reducing e_1 or e_2, we proceed as in the case for the (*pair*) rule. Otherwise, we have necessarily $e_1 = \mu f(s_1 \to t_1; \ldots; s_n \to t_n).\lambda x.e_0$, $e' = e_0[f := e_1; x := e_2]$ and e_2 is a value v_2. We have $\bigwedge_{i \in I} s_i \to t_i \leq s \to t$, where $I = \{1, \ldots, n\}$. According to Lemma 24, this means that $s \leq \bigvee_{i \in I} s_i$ and that for any non-empty $I' \subseteq I$ such that $s \not\leq \bigvee_{i \in I \setminus I'} s_i$, we have $\bigwedge_{i \in I'} t_i \leq t$. We take $I' = \{i \in I \mid \vdash v_2 : s_i\}$. This set is not empty. Indeed, since $\vdash v_2 : s$ and $s \leq \bigvee_{i \in I} s_i$, we have at least one i such that $\vdash v_2 : s_i$ (Lemma 39). Now, we claim that $s \not\leq \bigvee_{i \in I \setminus I'} s_i$. Otherwise, we would find some $i \notin I'$ such that $\vdash v_2 : s_i$, which contradicts the definition for I'. As a consequence, we get $\bigwedge_{i \in I'} t_i \leq t$. We claim that $\Gamma \vdash e' : \bigwedge_{i \in I'} t_i$ (which by subsumption yields $\Gamma \vdash e' : t$ i.e. the result). To prove our claim we show that for every $i \in I'$ we have $\Gamma \vdash e' : t_i$, which thanks to Lemma 31 yields our claim. So, let us consider any $i \in I'$, that is, any i such that $\vdash v_2 : s_i$. The abstraction e_1 is well-typed under Γ therefore in its derivation there is an instance of the (*abstr*) rule (possibly followed by several applications of the subsumption rule) which infers for e_1 a type t' under Γ. One of the premises of this rule is $(f : t'), (x : t_i), \Gamma \vdash e_0 : t_i$. We also have $\Gamma \vdash e_1 : t'$ and $\Gamma \vdash v_2 : s_i$ (Lemma 30), and thus $\Gamma \vdash e' : t_i$ (Lemma 33) as expected.

Rule (*abstr*): the expression e is an abstraction, and the reduction can only occur within its body. We proceed as in the case for the (*pair*) rule.

Rule (*case*): we have $e = (x = e_0 \in s \ ? \ e_1 \mid e_2)$. If the reduction occurs within one of the sub-expressions e_0,e_1,e_2, we proceed as in the case for the (*pair*) rule. Otherwise, the expression e_0 is necessarily a value v, and we have either $(\vdash v : s) \wedge (e' = e_1[x := v])$ or $(\vdash v : \neg s) \wedge (e' = e_2[x := v])$. Let us

consider for instance the first case. The typing rule gives: $\Gamma \vdash v : s_0$. Thanks to Lemma 31, we get $\Gamma \vdash v : s_0 \wedge s$. Because of Lemma 35, we know that $s_0 \wedge s \not\simeq \mathbb{0}$, that is $s_0 \not\leq \neg s$. So the typing rule (*case*) under consideration has a premise for e_1, namely $(x : s_0 \wedge s), \Gamma \vdash e_1 : t$. Lemma 33 gives $\Gamma \vdash e' : t$ as expected. □

And here is the proof of the progress property, Theorem 9 in Section 5.

Proof: We write $e \not\rightsquigarrow$ if e cannot be reduced ($\nexists e'.e \rightsquigarrow e'$). Suppose that $\vdash e : t$; we prove on induction on the derivation of $\vdash e : t$ that either e is a value or it can be reduced. We consider the last rule used in this derivation.
Rule (*subsum*): straightforward application of the induction hypothesis.
Rule (*var*): a variable cannot be well-typed in an empty environment. This case is thus impossible.
Rule (*const*): the expression e is a constant. It is thus a value.
Rule (*abstr*): the expression e is an abstraction which is well-typed under the empty environment. It is thus a value.
Rule (*proj*): we have $e = \pi_i(e_0)$, $t = t_i$, $\vdash e_0 : t_1 \times t_2$. If e_0 can be reduced to, say, e_0', then $e \rightsquigarrow \pi_i(e_0')$. Otherwise, if $e_0 \not\rightsquigarrow$, then by the induction hypothesis e_0 is a value. By Lemma 37, we get $e_0 = (v_1, v_2)$, and thus $e \rightsquigarrow v_i$.
Rule (*rnd*): we have $e = \texttt{rnd}(t)$ and thus $e \rightsquigarrow e'$ for any e' of type t (for instance, we can take for e' an expression of type $\mathbb{0}$, which exists).
Rule (*pair*): we have $e = (e_1, e_2)$, $t = t_1 \times t_2$, and $\vdash e_i : t_i$ for $i = 1..2$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by the induction hypothesis, we obtain that both e_1 and e_2 are values, and so is e.
Rule (*appl*): we have $e = e_1 e_2$, $\vdash e_1 : s \to t$ and $\vdash e_2 : s$. If one of the e_i can be reduced, then e can also be reduced. Otherwise, by the induction hypothesis, we obtain that both e_1 and e_2 are values. By Lemma 37, we get $e_1 = \mu f(s_1 \to t_1; \ldots; s_n \to t_n).\lambda x.e_0$. Then $e \rightsquigarrow e_0[f := e_1; x := e_2]$.
Rule (*case*): we have $e = (x = e_0 \in s \ ? \ e_1 \mid e_2)$. If e_0 can be reduced, then e can also be reduced. Otherwise, by the induction hypthesis, we obtain that e_0 is a value v. Because of Lemma 39, we have $\vdash v : s$ or $\vdash v : \neg s$, and thus $e \rightsquigarrow e_1[x := v]$ or $e \rightsquigarrow e_2[x := v]$. □

6.7 Construction of models

A naive idea to build a model would be to look for an interpretation domain D such that $D = \mathbb{E}D$. Of course such a set cannot exist, since the cardinality of $\mathbb{E}^{\mathbf{fun}}D$, and thus of $\mathbb{E}D$, is stricly larger than the cardinality of D. This cardinality problem can be avoided by considering only finite graphs to interpret functions.

For any set D, we write $\mathbb{E}_f D = \mathscr{C} + D^2 + \mathscr{P}_f(D \times D_\Omega)$ where $\mathscr{P}_f$ denotes the restriction of the powerset to finite subsets.

Definition 44 *A set-theoretic interpretation* $[\![\_]\!] : \mathscr{T} \to \mathscr{P}(D)$ *is* finitely extensional *if:*

1. $D = \mathbb{E}_f D$
2. $[\![a]\!] = \mathbb{E}(a) \cap D$ *for any atom* a.

Lemma 45 *If* $[\![\_]\!]$ *is a finitely extensional set-theoretic interpretation, then* $[\![t]\!] = \mathbb{E}(t) \cap D$ *for any type* t, *and* $[\![\tau]\!] = \mathbb{E}(\tau) \cap D$ *for any normal formal* τ.

Proof: Induction on t. □

The next lemma shows that taking finite sets as extensional models for functions does not change the subtyping relation between arrow types (compare it with Lemma 23).

Lemma 46 *Let* $(X_i)_{i\in P}$ *and* $(X_i)_{i\in N}$ *be two finite families of subsets of* D. *Then:*

$$\bigcap_{i\in P} \mathscr{P}_f(X_i) \subseteq \bigcup_{i\in N} \mathscr{P}_f(X_i) \iff \exists i_o \in N.\ \bigcap_{i\in P} X_i \subseteq X_{i_0}$$

Proof: The $\Leftarrow$ implication is straightforward. Let us prove $\Rightarrow$. We assume that any finite subset of $X = \bigcap_{i\in P} X_i$ is a subset of one of the X_{i_0} with $i_0 \in N$. We need to prove that the same holds for X itself. Otherwise, we could find for each $i_0 \in N$ an element $x_{i_0} \in X \backslash X_{i_0}$ and we would obtain a contradiction by considering the finite set $\{x_{i_0} \mid i_0 \in N\}$. □

Lemma 47 *Let* P, N *two finite sets of arrow types and* $[\![\_]\!]$ *an arbitratry set-theoretic interpretation. Then:*

$$\bigcap_{a\in P} \mathbb{E}(a) \subseteq \bigcup_{a\in N} \mathbb{E}(a) \iff \mathscr{P}_f(D \times D_\Omega) \cap \bigcap_{a\in P} \mathbb{E}(a) \subseteq \bigcup_{a\in N} \mathbb{E}(a)$$

(By convention $\bigcap_{a\in\varnothing} \mathbb{E}(a) = \mathscr{P}(D \times D_\Omega)$.*)*

Proof: Consequence of Lemmas 23, 46, and 22. □

It is, then, not surprising that finitely extensional interpretations are models.

Lemma 48 *Every finitely extensional interpretation is a model.*

Proof: Since $[\![\tau]\!] = \mathbb{E}(\tau) \cap D$, we need to prove that

$$\mathbb{E}(\tau) = \varnothing \iff \mathbb{E}(\tau) \cap D = \varnothing$$

for any normal form τ. We write:

$$\mathbb{E}(\tau) = \bigcup_{u \in U} \bigcup_{(P,N) \in \tau} \left(\mathbb{E}^u D \cap \bigcap_{a \in P} \mathbb{E}(a) \setminus \bigcup_{a \in N} \mathbb{E}(a) \right)$$

So we need to prove that for any $u \in U$ and (P, N) two finite sets of atoms, we have:

$$\mathbb{E}^u D \cap \bigcap_{a \in P} \mathbb{E}(a) \subseteq \bigcup_{a \in N} \mathbb{E}(a) \iff D \cap \mathbb{E}^u D \cap \bigcap_{a \in P} \mathbb{E}(a) \subseteq \bigcup_{a \in N} \mathbb{E}(a)$$

If $u \neq \mathbf{fun}$, then $\mathbb{E}^u D \subseteq D$, and the equivalence is thus trivial. The case $u = \mathbf{fun}$ comes from Lemma 47. □

6.8 A universal model

In this section, we define a structural and finitely extensional model and then show that it is universal and, in the next section, that the subtyping relation induced by this model is decidable.

We need to build a set D^0 such that $D^0 = \mathbb{E}_f D^0$, that is, a solution to the equation $D^0 = \mathscr{C} + D^0 \times D^0 + \mathscr{P}_f(D^0 \times D^0_\Omega)$. We will consider the *initial* solution to this equation. Concretely, we define D^0 as the set of finite terms generated by the production d of the following grammar (c ranges over elements of $\mathscr{C}$):

$$\begin{array}{lcl} d & ::= & c \mid (d, d) \mid \{(d, d'), \ldots, (d, d')\} \\ d' & ::= & d \mid \Omega \end{array}$$

Now, we need to define a set-theoretic interpretation $[\![\_]\!]^0 : \mathscr{T} \to \mathscr{P}(D^0)$ such that $[\![t]\!]^0 = \mathbb{E}(a)^0 \cap D^0$. Because of the inductive structure of elements of D^0, this equation actually defines the function $[\![\_]\!]^0$. To see this, we will define a binary predicate $(d : t)$ where $d \in D^0$ and $t \in \mathscr{T}$. The truth value of $(d : t)$ is defined by induction on the pair (d, t) ordered lexicographically, using the inductive structure for elements of D^0, and the induction principle we mentioned earlier for types. Here is the definition:

$$\begin{array}{lcll} (c : b) & = & c \in \mathbb{B}[\![b]\!] & \\ ((d_1, d_2) : t_1 \times t_2) & = & (d_1 : t_1) \wedge (d_2 : t_2) & \\ (\{(d_1, d'_1), \ldots, (d_n, d'_n)\} : t_1 \to t_2) & = & \forall i.\ (d_i : t_1) \Rightarrow (d'_i : t_2) & \\ (d : t_1 \vee t_2) & = & (d : t_1) \vee (d : t_2) & \\ (d : \neg t) & = & \neg (d : t) & \\ (d : t) & = & \texttt{false} & \text{otherwise} \end{array}$$

Now we define $[\![t]\!]^0 = \{d \in D^0 \mid (d : t)\}$. It is straightforward from this definition to see that $[\![\_]\!]^0$ is a set-theoretic interpretation and that it is structural (and thus well-founded). It is also clear that it is finitely extensional. It is thus a model. It remains to prove that this model is universal. This is a direct consequence of the next lemma.

Lemma 49 *If $\mathscr{S}^0 = \{\tau \mid [\![\tau]\!]^0 = \varnothing\}$ and $\mathscr{S}$ is a simulation, then $\mathscr{S} \subseteq \mathscr{S}^0$.*

Proof: Let $\mathscr{S}$ be a simulation. We need to prove that $\forall \tau \in \mathscr{S}.\ [\![\tau]\!]^0 = \varnothing$, that is:

$$\forall d \in D^0. \forall \tau \in \mathscr{S}.\ d \notin [\![\tau]\!]^0$$

We will prove this property by induction on $d \in D^0$. Let's take $d \in D^0$ and $\tau \in \mathscr{S}$. Since $\mathscr{S}$ is a simulation, we also have $\tau \in \mathbb{E}\mathscr{S}$, that is:

$$\forall u \in U. \forall (P,N) \in t.\ (P \subseteq \mathscr{A}_u \Rightarrow C_u^{P,N \cap \mathscr{A}_u})\} \tag{6}$$

where the conditions $C_u^{P,N}$ are as in Definition 25.
We need to prove that $d \notin [\![\tau]\!]^0$. The set $[\![\tau]\!]^0$ is equal to:

$$\bigcup_{(P,N)\in\tau} \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N} [\![a]\!]^0$$

We prove that d does not belong to any of the terms of this union. Let $(P,N) \in \tau$ and u be the kind of d (as for values, it is straightforward to associate a unique kind to each element of D^0). If $a \in \mathscr{A} \setminus \mathscr{A}_u$, then clearly $d \notin [\![a]\!]^0$. As a consequence, if $P \not\subseteq \mathscr{A}_u$, then $d \notin \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N} [\![a]\!]^0$. We now assume that $P \subseteq \mathscr{A}_u$. We can apply (6). We obtain that $C_u^{P,N\cap\mathscr{A}_u}$ holds. It remains to prove that:

$$d \notin \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N\cap\mathscr{A}_u} [\![a]\!]^0$$

$\underline{u = \mathbf{basic},\ d = c}$. The condition $C_u^{P,N\cap\mathscr{A}_u}$ is:

$$\mathscr{C} \cap \bigcap_{b\in P} \mathbb{B}[\![b]\!] \subseteq \bigcup_{b\in N} \mathbb{B}[\![b]\!]$$

As a consequence, we get:

$$d \notin \bigcap_{b\in P} \mathbb{B}[\![b]\!] \setminus \bigcup_{b\in N} \mathbb{B}[\![b]\!] = \bigcap_{a\in P} [\![a]\!]^0 \setminus \bigcup_{a\in N\cap\mathscr{A}_{\mathbf{basic}}} [\![a]\!]^0$$

$\underline{u = \mathbf{prod},\ d = (d_1, d_2)}$. The condition $C_u^{P,N\cap\mathscr{A}_u}$ is:

$$\forall N' \subseteq N \cap \mathscr{A}_{\mathbf{prod}}. \begin{cases} \mathscr{N}\left(\bigwedge_{t_1\times t_2\in P} t_1 \wedge \bigwedge_{t_1\times t_2\in N'} \neg t_1\right) \in \mathscr{S} \\ \qquad\qquad \vee \\ \mathscr{N}\left(\bigwedge_{t_1\times t_2\in P} t_2 \wedge \bigwedge_{t_1\times t_2\in N\setminus N'} \neg t_2\right) \in \mathscr{S} \end{cases}$$

For each N', we apply the induction hypothesis to d_1 and to d_2. We get:

$$d_1 \notin \left[\!\!\left[\bigwedge_{t_1 \times t_2 \in P} t_1 \wedge \bigwedge_{t_1 \times t_2 \in N'} \neg t_1 \right]\!\!\right]^0 \vee d_2 \notin \left[\!\!\left[\bigwedge_{t_1 \times t_2 \in P} t_2 \wedge \bigwedge_{t_1 \times t_2 \in N \setminus N'} \neg t_2 \right]\!\!\right]^0$$

That is:

$$d \notin \left(\bigcap_{t_1 \times t_2 \in P} [\![t_1]\!]^0 \setminus \bigcup_{t_1 \times t_2 \in N'} [\![t_1]\!]^0 \right) \times \left(\bigcap_{t_1 \times t_2 \in P} [\![t_2]\!]^0 \setminus \bigcup_{t_1 \times t_2 \in N \setminus N'} [\![t_2]\!]^0 \right)$$

According to Lemma 20 and to $[\![t_1]\!]^0 \times [\![t_2]\!]^0 = [\![t_1 \times t_2]\!]^0$, we thus get:

$$d \notin \bigcap_{a \in P} [\![a]\!]^0 \setminus \bigcup_{a \in N \cap \mathscr{A}_{\mathbf{prod}}} [\![a]\!]^0$$

$u = \mathbf{fun}$, $d = \{(d_1, d_1'), \ldots, (d_n, d_n')\}$. The condition $C_u^{P, N \cap \mathscr{A}_u}$ says that there exists $t_0 \to s_0 \in N$ such that, for all $P' \subseteq P$:

$$\mathscr{N}\left(t_0 \wedge \bigwedge_{t \to s \in P'} \neg t \right) \in \mathscr{S} \vee \begin{cases} P \neq P' \\ \mathscr{N}\left((\neg s_0) \wedge \bigwedge_{t \to s \in P \setminus P'} s \right) \in \mathscr{S} \end{cases}$$

Applying the induction hypothesis to the d_i and d_i' (note that if $d_i' = \Omega$, then $d_i' \notin [\![\tau]\!]^0$ is trivial for all τ):

$$d_i \notin \left[\!\!\left[t_0 \wedge \bigwedge_{t \to s \in P'} \neg t \right]\!\!\right]^0 \vee \begin{cases} P \neq P' \\ d_i' \notin \left[\!\!\left[(\neg s_0) \wedge \bigwedge_{t \to s \in P \setminus P'} s \right]\!\!\right]^0 \end{cases}$$

Let us first assume that $\forall i.\ (d_i \in [\![t_0]\!]^0 \Rightarrow d_i' \in [\![s_0]\!]^0)$. Then we have $d \in [\![t_0 \to s_0]\!]^0$. Otherwise, let us consider i such that $d_i \in [\![t_0]\!]^0$ and $d_i' \notin [\![s_0]\!]^0$. The formula above gives for any $P' \subseteq P$:

$$\left(d_i \in \bigcup_{t \to s \in P'} [\![t]\!]^0 \right) \vee \left(P' \neq P \wedge d_i' \in \{\Omega\} \cup \bigcup_{t \to s \in P \setminus P'} [\![\neg s]\!]^0 \right)$$

Let's take $P' = \{t \to s \in P \mid d_i \notin [\![t]\!]^0\}$. We have $d_i \notin \bigcup_{t \to s \in P'} [\![t]\!]^0$, and thus $P' \neq P$ and $d_i' \in \{\Omega\} \cup \bigcup_{t \to s \in P \setminus P'} [\![\neg s]\!]^0$. We can thus find $t \to s \in P \setminus P'$ such that $d_i' \notin [\![s]\!]^0$, and because $t \to s \notin P'$, we also have $d_i \in [\![t]\!]^0$. We have thus proved that $d \notin [\![t \to s]\!]^0$ for some $t \to s \in P$.
In both cases, we get:

$$d \notin \bigcap_{a \in P} [\![a]\!]^0 \setminus \bigcup_{a \in N \cap \mathscr{A}_{\mathbf{fun}}} [\![a]\!]^0$$

□

6.9 Subtyping decidability for the universal model

We will now focus on Theorem 15. Let $\leq_0$ denote the subtyping relation induced by the universal model $[\![\_]\!]^0$. We have $t_1 \leq_0 t_2 \iff [\![t_1 \backslash t_2]\!]^0 = \varnothing \iff [\![\mathscr{N}(t_1 \backslash t_2)]\!]^0 = \varnothing$. Therefore we need to show how to decide, for a given normal form τ_0, whether $[\![\tau_0]\!]^0 = \varnothing$ or not. Thanks to the Lemma above, we get: $[\![\tau_0]\!]^0 = \varnothing$ if and only if there exists a simulation $\mathscr{S}$ such that $\tau_0 \in \mathscr{S}$.

Actually, we can restrict our attention to a finite number of normal forms. Indeed, let us consider the set A of all the atoms that occur in τ_0 (including atoms nested in other atoms). Thanks to the regularity of types, this set A is finite. Write $\mathscr{N}(A)$ for the set of normal forms built only on top of these atoms, that is: $\mathscr{N}(A) = \mathscr{P}(\mathscr{P}(A) \times \mathscr{P}(A))$. This set is also finite, and looking at Definition 25, we see that an intersection of a simulation and $\mathscr{N}(A)$ is again a simulation. As a consequence, we get: $[\![\tau_0]\!]^0 = \varnothing$ if and only if there exists a simulation $\mathscr{S} \subseteq \mathscr{N}(A)$ such that $\tau_0 \in \mathscr{S}$. A naive algorithm can simply enumerate all the subset of $\mathscr{N}(A)$ which contains τ_0 and by applying Definition 25 check whether one of them is a simulation.

Of course, there exist better algorithms. For instance, we can interpret the definition of a simulation as saturation rules: the algorithm starts from the set $\{\tau_0\}$ and tries to saturate it until it obtains a simulation. Because of the disjunctions in the definition of a simulation, this algorithm needs to explore different branches. A branch cannot be infinite because the algorithm will only consider the normal forms in $\mathscr{N}(A)$ which is a finite set. There exists a simulation which contains τ_0 if and only if one of the branches succeeds in reaching a simulation. The Ph.D. thesis [14] describes two algorithms which improve over this simple saturation-based strategy.

6.10 Non-universal models

The interpretation domain D of a finitely extensional set-theoretic interpretation must be a solution to the equation $D = \mathbb{E}_f D$. In the previous section, we considered the initial solution to this equation and we obtained a universal model. In this section, we will build non-universal models by considering non-initial solutions to the equation $D = \mathbb{E}_f D$.

A first attempt could be to consider *infinite* (or maybe regular) terms generated by the following productions:

$$\begin{array}{lcl} d & ::= & c \mid (d,d) \mid \{(d,d'),\ldots,(d,d')\} \\ d' & ::= & d \mid \Omega \end{array}$$

But it is then impossible to build a finitely extensional interpretation on this domain D^∞. Indeed, if $[\![\_]\!]$ is such an interpretation, we consider the element $d \in D^\infty$ such that $d = (d,d)$ and the type t such that $t = (\neg t) \times (\neg t)$. Since $d \in D^\infty$ and $[\![t]\!] = \mathbb{E}(t) \cap D^\infty = (D^\infty \backslash [\![t]\!]) \times (D^\infty \backslash [\![t]\!])$, we have: $d \in [\![t]\!] \iff (d,d) \in (D^\infty \backslash [\![t]\!]) \times (D^\infty \backslash [\![t]\!]) \iff d \notin [\![t]\!]$. Contradiction.

So, we will build domains which are intermediate between D^0 and D^∞. We need to introduce some new notions.

For an arbitrary set X, we define $D^{[X]}$ as the set of finite terms generated by the production d below:

$$\begin{array}{lcl} d & ::= & x \mid c \mid (d,d) \mid \{(d,d'),\ldots,(d,d')\} \\ d' & ::= & d \mid \Omega \end{array}$$

where x ranges over elements of X. In other words, $D^{[X]}$ is the initial solution D to the equation $D = X + \mathscr{C} + D^2 + \mathscr{P}_f(D \times D_\Omega)$. We define the predicate $\Delta \vdash d : t$ for $d \in D^{[X]}$, $t \in \mathscr{T}$, $\Delta \in \mathscr{P}(\mathscr{T})^X$ by induction on the structure of d:

$$\begin{array}{lcll} (\Delta \vdash d : t_1 \vee t_2) & = & (\Delta \vdash d : t_1) \vee (\Delta \vdash d : t_2) & \\ (\Delta \vdash d : \neg t) & = & \neg(\Delta \vdash d : t) & \\ (\Delta \vdash c : b) & = & c \in \mathbb{B}[\![b]\!] & \\ (\Delta \vdash (d_1,d_2) : t_1 \times t_2) & = & (\Delta \vdash d_1 : t_1) \wedge (\Delta \vdash d_2 : t_2) & \\ (\Delta \vdash \{(d_1,d'_1),\ldots,(d_n,d'_n)\} : t_1 \to t_2) & = & \forall i.\ (\Delta \vdash d_i : t_1) \Rightarrow (\Delta \vdash d'_i : t_2) & \\ (\Delta \vdash x : a) & = & a \in \Delta(x) & \\ (\Delta \vdash d : t) & = & \texttt{false} & \text{otherwise} \end{array}$$

A congruence on $D^{[X]}$ is an equivalence relation $\equiv$ such that $(d_1^1 \equiv d_1^2 \wedge d_2^1 \equiv d_2^2) \Rightarrow (d_1^1, d_2^1) \equiv (d_1^2, d_2^2)$ and $(\forall i. d_i^1 \equiv d_i^2 \wedge d_i'^1 \equiv d_i'^2) \Rightarrow \{(d_1^1, d_1'^1), \ldots\} \equiv \{(d_1^2, d_1'^2), \ldots\}$. If for all x, we choose an element $d^x \in \mathbb{E}_f(D^{[X]}) = D^{[X]} \backslash X$ and if we consider the smallest congruence $\equiv$ such that $\forall x \in X. x \equiv d^x$, then the quotient $D_\equiv^{[X]} = D^{[X]}/\equiv$ is such that $\mathbb{E}_f(D_\equiv^{[X]}) = D_\equiv^{[X]}$ (modulo an implicit bijection). Let's choose some $\Delta \in \mathscr{P}(\mathscr{T})^X$. We require the predicate $(\Delta \vdash d : t)$ to be invariant under $\equiv$, that is: $d^1 \equiv d^2 \Rightarrow ((\Delta \vdash d^1 : t) \iff (\Delta \vdash d^2 : t))$. This is the case if and only if $\forall x.(\Delta \vdash x : t) \iff (\Delta \vdash d^x : t)$, that is, if and only if:

$$(*) \quad \forall x \in X.\ \Delta(x) = \{t \mid \Delta \vdash d^x : t\}$$

When this property holds, we can define $[\![\_]\!]_\Delta : \mathscr{T} \to \mathscr{P}(D_\equiv^{[X]})$ by $[\![t]\!]_\Delta = \{[d]_\equiv \mid (\Delta \vdash d : t)\}$, where $[d]_\equiv$ denotes the equivalence class of d modulo $\equiv$. This defines a finitely extensional set-theoretic interpretation (and thus a model).

Of course, the difficulty is now to choose X, the d^x and Δ such that $(*)$ holds. Let us consider the case where $X = \mathbb{Z}$, and each $d^k, k \in \mathbb{Z}$ is defined using only d^{k-1} in a uniform way. Formally, we consider a fixed element $\delta \in D^{\{\bullet\}}$ such that $\delta \neq \bullet$ and we define $d^k = \delta[\bullet := k-1]$ (that is, the element of $D^\mathbb{Z}$ obtained by substituting $\bullet$ by $k-1$ in δ). If $\Delta \in \mathscr{P}(\mathscr{T})^\mathbb{Z}$, then $\Delta \vdash d^k : t$ is equivalent to $\Delta \vdash \delta[\bullet := k-1] : t$, and an induction on the structure of δ shows that this is equivalent to $(\bullet \mapsto \Delta_{k-1}) \vdash \delta : t$. If we define the operator $F : \mathscr{P}(\mathscr{T}) \to \mathscr{P}(\mathscr{T})$ by $F(T) = \{t \mid (\bullet \mapsto T) \vdash \delta : t\}$, then the condition $(*)$ can be rewritten as:

$$\forall k \in \mathbb{Z}.\ \Delta_k = F(\Delta_{k-1})$$

Building such a sequence is not straightforward. We will rely on a technical lemma.

Lemma 50 *Let A be a finite set, $f : A \to A$, and $a^0 \in A$. There exists a unique periodic sequence $(a_k)_{k\in\mathbb{Z}} \in A^{\mathbb{Z}}$ such that:*

$$\exists n_0 \in \mathbb{N}.\forall k \geq n_0.a_k = f^k(a^0)$$

(where f^n denotes the n-th iterated composition of f with itself). This sequence is such that:

$$\forall k.\ a_{k+1} = f(a_k)$$

Proof: We consider the sequence $(a^n)_{n\in\mathbb{N}}$ defined by $a^n = f^n(a^0)$. Since A is finite, this sequence cannot be injective. We can find $n_0 < n_1$ such that $a^{n_0} = a^{n_1}$. A recurrence gives $a^n = a^{n+(n_1-n_0)}$ for any $n \geq n_0$: the sequence $(a^n)_{n\in\mathbb{N}}$ is ultimately periodic. As a consequence, there exists a unique sequence $(a_k)_{k\in\mathbb{Z}}$ which coincides ultimately with $(a^n)_{n\in\mathbb{N}}$.
Clearly, the property $a_{k+1} = f(a_k)$ holds for k large enough, and because $(a_k)_{k\in\mathbb{Z}}$ is periodic, it holds for any k. □

Theorem 51 *Let T^0 be a set of types. There exists a sequence $(\Delta_k)_{k\in\mathbb{Z}}$ such that:*

- $\forall k \in \mathbb{Z}.\Delta_{k+1} = F(\Delta_k)$
- *For any type t, the sequence of the truth values of $(t \in \Delta_k)_{k\in\mathbb{Z}}$ is periodic and $\exists n_0 \in \mathbb{N}.\forall k \geq n_0.(t \in \Delta_k \iff t \in F^k(T^0))$*

Proof: Since the set $\mathscr{P}(\mathscr{T})$ is not finite, we cannot use the lemma directly. The regularity of types will come to the rescue. We define a cone as a *finite* set of types which is closed under subterms decomposition (that is, if the set contains a type, it also contains all its subterms). Any type belongs to some cone because a type is a regular term. For a cone C, we can define the function $F_C : \mathscr{P}(C) \to \mathscr{P}(C)$ by $F_C(T) = F(T) \cap C$. We can apply the lemma to this function, because $\mathscr{P}(C)$ is finite. We write $(T_k^C)_{k\in\mathbb{Z}}$ for the sequence we obtain. Now, we observe on the definition of the $\vdash$ predicate that for $t \in C$, the assertion $(\bullet \mapsto T) \vdash \delta : t$ holds if and only if $(\bullet \mapsto (T \cap C)) \vdash \delta : t$ holds. This gives immediatly the following property:

$$\forall T \subseteq \mathscr{T}.\ C \cap F(T \cap C) = C \cap F(T)$$

From that, a recurrence gives $F_C^n(T^0) = F^n(T^0) \cap C$. So, for $t \in C$, we have $t \in T_k^C \iff t \in F^k(T_0)$ when k is large enough. Since the sequence $(t \in T_k^C)_{k\in\mathbb{Z}}$ is periodic, it does not depend on the choice of the cone C which contains t. We can thus define Δ_k as the set of types t such that $t \in T_k^C$ for some/any cone C that contains t. We have $T_k^C = \Delta_k \cap C$. It remains to check that $\Delta_{k+1} = F(\Delta_k)$ for all k. Let t be a type and C a cone which contains t. We have $t \in \Delta_{k+1} \iff t \in T_{k+1}^C$ and according to the lemma, we have $T_{k+1}^C = F(T_k^C) \cap C = F(\Delta_k) \cap C$. So: $t \in \Delta_{k+1} \iff t \in F(\Delta_k)$. Since this property holds for an arbitrary t, we get $\Delta_{k+1} = F(\Delta_k)$ as expected. □

We will give two examples of constructions based on this theorem. First, we will build a model which is not well-founded. In a well-founded model, the recursive type $t_0 = t_0 \times t_0$ is empty. We will build a model where this type is not empty. We take $\delta = (\bullet, \bullet)$ and we build $(\Delta_k)_{k\in\mathbb{Z}}$ as given by the theorem. We thus get a finitely extensional set-theoretic interpretation $[\![\_]\!]_\Delta : \mathscr{T} \to \mathscr{P}(D_{\equiv}^{\mathbb{Z}})$. For any set of types T, we have $t_0 \in F(T) \iff (\bullet \mapsto T) \vdash \delta : t_0 \iff (\bullet \mapsto T) \vdash (\bullet, \bullet) : t_0 \times t_0 \iff (\bullet \mapsto T) \vdash \bullet : t_0 \iff t_0 \in T$. So if we choose T^0 such that $t_0 \in T^0$, we have $t_0 \in \Delta_k$ for all k, from which we conclude that $[\![t_0]\!]_\Delta$ contains the $[k]_\equiv$ for $k \in \mathbb{Z}$. In particular, it is not empty. To better understand our construction, we can consider the type $t_1 = (\neg t_1) \times (\neg t_1)$. We find that $t_1 \in F(T) \iff t_1 \notin T$ and we deduce that $[\![t_1]\!]_\Delta$ contains the $[k]_\equiv$ for all even $k \in \mathbb{Z}$ (if $t_1 \in T^0$) or for all $k \in \mathbb{Z}$ (if $t_1 \notin T^0$). For more complex recursive types, we might see other periods that 2.

Now, we will build a structural (and thus well-founded) model which is not universal. We consider the recursive type $t_0 = (\mathbb{0} \to \mathbb{0}) \setminus (t_0 \to \mathbb{0})$. If $[\![\_]\!]$ is a finitely extensional set-theoretic interpretation, a simple computation gives:

$$[\![t_0]\!] = \{(d_i, d'_i) \mid \exists i.\ d_i \in [\![t_0]\!]\}$$

In particular, this set is empty for the universal model built in the previous section (because its elements are finite trees). We take $\delta = \{(\bullet, \Omega)\}$ and we proceed as above, with the following computation: $t_0 \in F(T) \iff (\bullet \mapsto T) \vdash \delta : t_0 \iff (\bullet \mapsto T) \vdash \{(\bullet, \Omega)\} : (\mathbb{0} \to \mathbb{0}) \setminus (t_0 \times \mathbb{0}) \iff (\bullet \mapsto T) \vdash \bullet : t_0 \iff t_0 \in T$. We conclude that the model $[\![\_]\!]_\Delta$ is not empty. It remains to see that it is structural. The decomposition relation $\triangleright$ is defined by $([d_1]_\equiv, [d_2]_\equiv) \triangleright [d_i]_\equiv$. Because of the definition of δ, if $[d]_\equiv \triangleright [d']_\equiv$, then d must be a pair (d_1, d_2) in $D^Z \times D^Z$. As a consequence, the relation $\triangleright$ is noetherian.

6.11 Towards type-checking

In this section, we introduce notions that will be useful for deriving a type-checking algorithm. We also give the proof of Theorem 10 (local exactness of the application rule). The existence results in this section are *effective* (viz. it is possible to compute the objets whose existence is asserted) provided that the subtyping relation is decidable.

Lemma 52 *Let t be a type such that $t \leq \mathbb{1} \times \mathbb{1}$. There exists a finite set of pairs of types $\pi(t) \in \mathscr{P}_f(\mathscr{T}^2)$ such that:*

- $t \simeq \bigvee_{(t_1,t_2)\in\pi(t)} t_1 \times t_2$
- $\forall (t_1, t_2) \in \pi(t).\ t_1 \not\simeq \mathbb{0} \wedge t_2 \not\simeq \mathbb{0}$

Proof: We can write:

$$t \simeq \bigvee_{(P,N)\in\mathcal{N}(t) \mid P\subseteq\mathscr{A}_{\mathbf{prod}}} (\mathbb{1}\times\mathbb{1})\wedge\bigwedge_{a\in P} a\setminus\bigvee_{a\in N\cap\mathscr{A}_{\mathbf{prod}}} a$$

Using Lemma 20, we can rewrite any intersection of product types and complement of product types as a union of product types $P' \subseteq \mathscr{A}_{\mathbf{prod}}$:

$$t \simeq \bigvee_{a\in P'} a$$

We simply define $\pi(t)$ as $\{(t_1,t_2) \mid t_1\times t_2 \in P' \wedge t_1 \not\simeq \mathbb{0} \wedge t_2 \not\simeq \mathbb{0}\}$. □

Lemma 53 *Let t be a type such that $t \leq \mathbb{0}\to\mathbb{1}$. Then there exists a finite set of pairs of types $\rho(t) \in \mathscr{P}_f(\mathscr{T}^2)$ and a type $\mathtt{Dom}(t)$ such that:*

$$\forall t_1,t_2.\ (t \leq t_1\to t_2) \iff \begin{cases} t_1 \leq \mathtt{Dom}(t) \\ \forall (s_1,s_2)\in\rho(t).\ (t_1\leq s_1)\vee(s_2\leq t_2) \end{cases}$$

Proof: We can write:

$$t \simeq \bigvee_{(P,N)\in\mathcal{N}(t) \mid P\subseteq\mathscr{A}_{\mathbf{fun}}} (\mathbb{0}\to\mathbb{1})\wedge\bigwedge_{a\in P} a\setminus\bigvee_{a\in N\cap\mathscr{A}_{\mathbf{fun}}} a$$

Clearly, the Lemma is true for $t \simeq \mathbb{0}$ (with $\mathtt{Dom}(t) = \mathbb{1}$ and $\rho(t) = \varnothing$), and if it holds for t and t', then it also holds for $t\vee t'$ (with $\mathtt{Dom}(t\vee t') = \mathtt{Dom}(t)\wedge\mathtt{Dom}(t')$ and $\rho(t\vee t') = \rho(t)\cup\rho(t')$). We can thus assume with loss of generality that t has the form:

$$t = \bigwedge_{a\in P} a\setminus\bigvee_{a\in N} a$$

with $P,N \subseteq \mathscr{A}_{\mathbf{fun}}$, $P \neq \varnothing$, and $t \not\simeq \mathbb{0}$. We conclude easily with Lemma 24. □

Corollary 54 *Let t and t_1 be two types. If $t \leq t_1\to\mathbb{1}$, then $t \leq t_1\to t_2$ has a smallest solution t_2 which we write $t \bullet t_1$.*

Proof: Since $t \leq t_1\to\mathbb{1}$, we have $t_1 \leq \mathtt{Dom}(t)$. The assertion $t \leq t_1\to t_2$ is thus equivalent to:

$$\forall (s_1,s_2)\in\rho(t).\ (t_1\leq s_1)\vee(s_2\leq t_2)$$

that is:

$$\left(\bigvee_{(s_1,s_2)\in\rho(t) \mid (t_1\not\leq s_1)} s_2\right) \leq t_2$$

We write $t\bullet t_1$ for the left-hand side of this equation. □

We can now prove Theorem 10.

Proof: Let t, t_1 be two types such that $t \leq t_1 \to \mathbb{1}$. Clearly, if $\vdash v_f : t$ and $\vdash v_x : t_1$, then $\vdash v_f v_x : t \bullet t_1$, and thus, subject reduction gives $\vdash v : t \bullet t_1$ if $v_f v_x \overset{\star}{\rightsquigarrow} v$.
Let's prove the opposite implication:

$$\forall v.\ \vdash v : t \bullet t_1 \Rightarrow \exists v_f, v_x.\ (v_f v_x \overset{\star}{\rightsquigarrow} v) \wedge (\vdash v_f : t) \wedge (\vdash v_x : t_1)$$

This property is clearly true for $t \simeq \mathbb{0}$, and if it is true for t and t', then it is true for $t \vee t'$ (because $\mathbb{0} \bullet t_1 \simeq \mathbb{0}$ and $(t \vee t') \bullet t_1 \simeq (t \bullet t_1) \vee (t' \bullet t_1)$). We can thus assume, as in the proof of Lemma 53, that t has the form:

$$t = \bigwedge_{a \in P} a \setminus \bigvee_{a \in N} a$$

with $P, N \subseteq \mathscr{A}_{\mathbf{fun}}$, $P \neq \varnothing$, and $t \not\simeq \mathbb{0}$. Lemma 24 gives:

$$t \bullet t_1 = \bigvee_{P' \subseteq P \mid t_1 \not\leq \bigvee_{t'_1 \to t'_2 \in P'} t'_1} \left(\bigwedge_{t'_1 \to t'_2 \in P \setminus P'} t'_2 \right)$$

and

$$t_1 \leq \bigvee_{t'_1 \to t'_2 \in P} t'_1$$

Let v be a value of type $t \bullet t_1$. We can find $P' \subseteq P$ such that $t_1 \not\leq \bigvee_{t'_1 \to t'_2 \in P'} t'_1$ and $\vdash v : \bigwedge_{t'_1 \to t'_2 \in P \setminus P'} t'_2$. Let v_x be a value of type $t_1 \setminus \bigvee_{t'_1 \to t'_2 \in P'} t'_1$ and v_f the abstraction

$$\mu f(P).\lambda x.\ (y = x \in \bigvee_{t'_1 \to t'_2 \in P'} t'_1\ ?\ fy \mid v)$$

It is then easy to check that $\vdash v_f : t$ and $v_f v_x \overset{\star}{\rightsquigarrow} v$. □

6.12 Type-checking algorithm

In this section, we assume that the subtyping relation $\leq$ is decidable and we give a type-checking algorithm for our type system.

The key difficulty to overcome is that the set of types t such that $\Gamma \vdash e : t$, for a given environment Γ and a given expression e has no smallest element in general. Indeed, consider the case where e is a well-typed abstraction. The (*abstr*) rule allows us to choose an arbitrary number of arrow types.

We will thus introduce a new syntactic category, called type scheme to denote such sets of types. The syntax for type schemes is given by the following productions:

$$\begin{array}{rcll} \mathbb{t} & ::= & t & t \in \mathscr{T} \\ & | & [t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n] & n \geq 1; t_i, s_i \in \mathscr{T} \\ & | & \mathbb{t}_1 \otimes \mathbb{t}_2 & \\ & | & \mathbb{t}_1 \oslash \mathbb{t}_2 & \\ & | & \Omega & \end{array}$$

We will write $[t_i \rightarrow s_i]_{i=1..n}$ for $[t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n]$. We define the function $\{\_\}$ which maps schemes to sets of types:

$$\begin{array}{lcl} \{t\} & = & \{s \mid t \leq s\} \\ \{[t_i \rightarrow s_i]_{i=1..n}\} & = & \{s \mid \exists s_0 = \bigwedge_{i=1..n} (t_i \rightarrow s_i) \wedge \bigwedge_{j=1..m} \neg(t'_j \rightarrow s'_j).\ \mathbb{0} \not\simeq s_0 \leq s\} \\ \{\mathbb{t}_1 \otimes \mathbb{t}_2\} & = & \{s \mid \exists t_1 \in \{\mathbb{t}_1\}, t_2 \in \{\mathbb{t}_2\}.\ t_1 \times t_2 \leq s\} \\ \{\mathbb{t}_1 \oslash \mathbb{t}_2\} & = & \{s \mid \exists t_1 \in \{\mathbb{t}_1\}, t_2 \in \{\mathbb{t}_2\}.\ t_1 \vee t_2 \leq s\} \\ \{\Omega\} & = & \varnothing \end{array}$$

Lemma 55 *Let $\mathbb{t}$ be a type schema. Then $\{\mathbb{t}\} = \varnothing$ if and only if Ω appears in $\mathbb{t}$. Moreover, $\{\mathbb{t}\}$ is closed under subsumption ($t \in \{\mathbb{t}\} \wedge t \leq t' \Rightarrow t' \in \{\mathbb{t}\}$) and intersection ($t \in \{\mathbb{t}\} \wedge t' \in \{\mathbb{t}\} \Rightarrow t \wedge t' \in \{t\}$).*

Proof: Straightforward induction of the structure of $\mathbb{t}$. □

Lemma 56 *Let $\mathbb{t}$ be a type scheme and t_0 a type. We can compute a type scheme, written $t_0 \oslash \mathbb{t}$, such that:*

$$\{t_0 \oslash \mathbb{t}\} = \{s \mid \exists t \in \{\mathbb{t}\}.\ t_0 \wedge t \leq s\}$$

Proof: We define $t_0 \oslash \mathbb{t}$ by induction on $\mathbb{t}$. If $\mathbb{t}$ is a type t, we take $t_0 \oslash \mathbb{t} = t_0 \wedge t$. If $\mathbb{t}$ is a union $\mathbb{t}_1 \vee \mathbb{t}_2$, we distribute: $t_0 \oslash \mathbb{t} = (t_0 \oslash \mathbb{t}_1) \oslash (t_0 \oslash \mathbb{t}_2)$. If $\mathbb{t}$ is Ω, or if $\{\mathbb{t}\} = \varnothing$, we take $t_0 \oslash \mathbb{t} = \Omega$. For the two remaining cases, we assume that $\mathbb{t} \neq \varnothing$, and we observe that:

$$t_0 \simeq \bigvee_{(P,N) \in \mathscr{N}(t)} \bigwedge_{a \in P} a \wedge \bigwedge_{a \in N} \neg a$$

We can thus see t_0 as a boolean combination built with $\mathbb{0}$, $\mathbb{1}$, $\vee$, $\wedge$, atoms and complement of atoms. For $t_0 \simeq \mathbb{0}$, we take $t_0 \oslash \mathbb{t} = \mathbb{0}$. For $t_0 \simeq \mathbb{1}$, we take $t_0 \oslash \mathbb{t} = \mathbb{t}$. For $t_0 \simeq t_1 \vee t_2$, we take $t_0 \oslash \mathbb{t} = (t_1 \oslash \mathbb{t}) \oslash (t_2 \oslash \mathbb{t})$. For $t_0 \simeq t_1 \wedge t_2$, we take $t_0 \oslash \mathbb{t} = t_1 \oslash (t_2 \oslash \mathbb{t})$. It remains to deal with the case of an atom or a complement of an atom.
For the case $\mathbb{t} = \mathbb{t}_1 \otimes \mathbb{t}_2$, we take:

$$(t_1 \times t_2) \oslash (\mathbb{t}_1 \otimes \mathbb{t}_2) = (t_1 \oslash \mathbb{t}_1) \otimes (t_2 \oslash \mathbb{t}_2)$$

$$\neg(t_1 \times t_2) \owedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = ((\neg t_1 \owedge \mathbb{t}_1) \otimes \mathbb{t}_2) \ovee (\mathbb{t}_1 \otimes (\neg t_2 \owedge \mathbb{t}_2))$$

and if $a \in \mathscr{A} \backslash \mathscr{A}_{\mathbf{prod}}$:

$$a \owedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = \mathbb{0}$$

$$\neg a \owedge (\mathbb{t}_1 \otimes \mathbb{t}_2) = (\mathbb{t}_1 \otimes \mathbb{t}_2)$$

For the case $\mathbb{t} = [t_i \rightarrow s_i]_{i=1..n}$, we take:

$$(t \rightarrow s) \owedge [t_i \rightarrow s_i]_{i=1..n} = \begin{cases} [t_i \rightarrow s_i]_{i=1..n} & \text{si } \bigwedge_{i=1..n} t_i \rightarrow s_i \leq t \rightarrow s \\ \mathbb{0} & \text{si } \bigwedge_{i=1..n} t_i \rightarrow s_i \not\leq t \rightarrow s \end{cases}$$

$$\neg(t \rightarrow s) \owedge [t_i \rightarrow s_i]_{i=1..n} = \begin{cases} \mathbb{0} & \text{si } \bigwedge_{i=1..n} t_i \rightarrow s_i \leq t \rightarrow s \\ [t_i \rightarrow s_i]_{i=1..n} & \text{si } \bigwedge_{i=1..n} t_i \rightarrow s_i \not\leq t \rightarrow s \end{cases}$$

and if $a \in \mathscr{A} \backslash \mathscr{A}_{\mathbf{fun}}$:

$$a \owedge [t_i \rightarrow s_i]_{i=1..n} = \mathbb{0}$$

$$\neg a \owedge [t_i \rightarrow s_i]_{i=1..n} = [t_i \rightarrow s_i]_{i=1..n}$$

□

Lemma 57 *Let $\mathbb{t}$ be a type scheme and t a type. We can decide the assertion $t \in \{\mathbb{t}\}$, which we also write $\mathbb{t} \leq t$.*

Proof: First, we make the observation that $t \in \{\mathbb{t}\}$ if and only if $\mathbb{0} \in \{(\neg t) \owedge \mathbb{t}\}$. Indeed: $\mathbb{0} \in \{(\neg t) \owedge \mathbb{t}\} \iff \exists s \in \{\mathbb{t}\}.\ (\neg t) \wedge s \leq \mathbb{0} \iff \exists s \in \{\mathbb{t}\}.\ s \leq t \iff t \in \{\mathbb{t}\}$. As a consequence, we only need to deal with the case $t = \mathbb{0}$. If $\{\mathbb{t}\} = \varnothing$, then $\mathbb{0} \in \{\mathbb{t}\}$ does not hold. Otherwise, we conclude by induction over the structure of $\mathbb{t}$:

$$\begin{array}{lcl} \mathbb{0} \in \{t\} & \iff & t \simeq \mathbb{0} \\ \mathbb{0} \notin \{[t_i \rightarrow s_i]_{i=1..n}\} & & \\ \mathbb{0} \in \{\mathbb{t}_1 \otimes \mathbb{t}_2\} & \iff & (\mathbb{0} \in \{\mathbb{t}_1\}) \vee (\mathbb{0} \in \{\mathbb{t}_2\}) \\ \mathbb{0} \in \{\mathbb{t}_1 \ovee \mathbb{t}_2\} & \iff & (\mathbb{0} \in \{\mathbb{t}_1\}) \wedge (\mathbb{0} \in \{\mathbb{t}_2\}) \\ \mathbb{0} \notin \{\Omega\} & & \end{array}$$

□

Lemma 58 *Let $\mathbb{t}$ by a type scheme and $i \in \{1, 2\}$. We can compute a type scheme $\pi_i(\mathbb{t})$ such that*

$$\pi_i(\mathbb{t}) = \{s \mid \exists t_1 \times t_2 \in \{\mathbb{t}\}.t_i \leq s\}$$

Proof: Let's take for instance $i = 1$. Note that $\exists t_1 \times t_2 \in \{\!|\mathbb{t}|\!\}.t_1 \leq s$ is equivalent to $s \times \mathbb{1} \in \{\!|\mathbb{t}|\!\}$.
If $\mathbb{t} \not\leq \mathbb{1} \times \mathbb{1}$, then we take $\{\!|\pi_1(\mathbb{t})|\!\} = \Omega$. Otherwise, we proceed by induction over the structure of $\mathbb{t}$. For $\mathbb{t} = \mathbb{t}_1 \vee \mathbb{t}_2$, we take $\pi_1(\mathbb{t}) = \pi_1(\mathbb{t}_1) \vee \pi_1(\mathbb{t}_2)$. For $\mathbb{t} = \mathbb{t}_1 \otimes \mathbb{t}_2$, we take $\pi_1(\mathbb{t}) = \mathbb{t}_1$. For $\mathbb{t} = t$, we take $\pi_1(\mathbb{t}) = \bigvee_{(t_1,t_2)\in\pi(t)} t_1$. The other cases are impossible. □

Lemma 59 *Let $\mathbb{t}$ and $\mathbb{t}_1$ be two type schemes. We can compute a type scheme $\mathbb{t} \bullet \mathbb{t}_1$ such that*

$$\{\!|\mathbb{t} \bullet \mathbb{t}_1|\!\} = \{s \mid \exists t_1 \to t_2 \in \{\!|\mathbb{t}|\!\}.t_1 \in \{\!|\mathbb{t}_1|\!\} \wedge t_2 \leq s\}$$

Proof: We proceed by induction over the structure of $\mathbb{t}$. For $\mathbb{t} = \mathbb{t}^1 \vee \mathbb{t}^2$, we take $\mathbb{t} \bullet \mathbb{t}_1 = \mathbb{t}^1 \bullet \mathbb{t}_1 \vee \mathbb{t}^2 \bullet \mathbb{t}_1$. For $\mathbb{t} = \mathbb{t}^1 \otimes \mathbb{t}^2$ or $\mathbb{t} = \Omega$, we take $\mathbb{t} \bullet \mathbb{t}_1 = \Omega$. For $\mathbb{t} = [t'_i \to s'_i]_{i=1..n}$, we take $\mathbb{t} \bullet \mathbb{t}_1 = (\bigwedge_{i=1..n}(t'_i \to s'_i)) \bullet \mathbb{t}_1$, so the only remaining case if $\mathbb{t} = t$. We observe that $\exists t_1 \to t_2 \in \{\!|\mathbb{t}|\!\}.t_1 \in \{\!|\mathbb{t}_1|\!\} \wedge t_2 \leq s$ is equivalent to $\exists t_1 \in \{\!|\mathbb{t}_1|\!\}.t \leq t_1 \to s$. According to Lemma 53, this is equivalent to: $\exists t_1 \in \{\!|\mathbb{t}_1|\!\}.t_1 \leq \mathtt{Dom}(t) \wedge \forall (s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq s)$. We claim that this is equivalent to $\mathbb{t}_1 \leq \mathtt{Dom}(t) \wedge \forall (s_1, s_2) \in \rho(t).\ (\mathbb{t}_1 \leq s_1) \vee (s_2 \leq s)$. The $\Rightarrow$ implication is immediate. Let us check the $\Leftarrow$ implication. For every $(s_1, s_2) \in \rho(t)$ such that $s_2 \not\leq s$, we have $\mathbb{t}_1 \leq s_1$ and it is thus possible to find a type $t'_1 \in \{\!|\mathbb{t}_1|\!\}$ such that $t'_1 \leq s_1$. We define t_1 as the intersection of all these t'_1 and of $\mathtt{Dom}(t)$, and we thus have $t_1 \in \{\!|\mathbb{t}_1|\!\} \wedge t_1 \leq \mathtt{Dom}(t) \wedge \forall (s_1, s_2) \in \rho(t).\ (t_1 \leq s_1) \vee (s_2 \leq s)$. To conclude, we define $t \bullet \mathbb{t}_1$ as Ω if $\mathbb{t}_1 \not\leq \mathtt{Dom}(t)$, and otherwise as:

$$\bigvee_{(s_1,s_2)\in\rho(t).(\mathbb{t}_1 \not\leq s_1)} s_2$$

□

We can now describe a type-checking algorithm. We define a scheme environment as a finite mapping Γ from variables to type schemes such that $\{\!|\Gamma(x)|\!\} \neq \varnothing$ for every x in the domain of Γ. The type-checking algorithm is formalized as a total function which maps a scheme environment Γ and an expression e to a scheme written $\Gamma[e]$. This function is defined by induction on the structure of e by the

following equations:

$$\begin{cases}
\mathbb{\Gamma}[c] = b_c \\
\mathbb{\Gamma}[(e_1, e_2)] = \mathbb{\Gamma}[e_1] \otimes \mathbb{\Gamma}[e_2] \\
\mathbb{\Gamma}[\mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e] = \begin{cases} \mathbb{t} & \text{if } \forall i = 1..n.\ \mathbb{s}_i \leq s_i \\ \Omega & \text{otherwise} \end{cases} \\
\qquad\qquad \text{where } \begin{cases} \mathbb{t} = [t_i \rightarrow s_i]_{i=1..n} \\ \mathbb{s}_i = ((f : \mathbb{t}), (x : t_i), \mathbb{\Gamma})[e] \quad (i = 1..n) \end{cases} \\
\mathbb{\Gamma}[x] = \begin{cases} \mathbb{\Gamma}(x) & \text{if } \mathbb{\Gamma}(x) \text{ is defined} \\ \Omega & \text{otherwise} \end{cases} \\
\mathbb{\Gamma}[\pi_i(e)] = \pi_i(\mathbb{\Gamma}[e]) \\
\mathbb{\Gamma}[e_1 e_2] = \mathbb{\Gamma}[e_1] \bullet \mathbb{\Gamma}[e_2] \\
\mathbb{\Gamma}[(x = e \in t\ ?\ e_1 | e_2)] = \mathbb{s}_1 \mathbin{\mathbb{\vee}} \mathbb{s}_2 \\
\qquad \text{where } \begin{cases} \mathbb{t}_0 = \mathbb{\Gamma}[e] \\ \mathbb{t}_1 = t \mathbin{\mathbb{\wedge}} \mathbb{t}_0 \\ \mathbb{t}_2 = (\neg t) \mathbin{\mathbb{\wedge}} \mathbb{t}_0 \\ \mathbb{s}_i = \begin{cases} ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i] & \text{if } \mathbb{t}_i \not\leq \mathbb{0}, \{\mathbb{t}_i\} \neq \varnothing \\ \mathbb{0} & \text{if } \mathbb{t}_i \leq \mathbb{0} \\ \Omega & \text{if } \{\mathbb{t}_i\} = \varnothing \end{cases} \quad (i = 1..2) \end{cases}
\end{cases}$$

We are now going to prove soundness and completeness of the algorithm. If $\mathbb{\Gamma}$ is a scheme environment and Γ is a typing environment, we write $\mathbb{\Gamma} \leq \Gamma$ when $\mathbb{\Gamma}$ and Γ have the same domain and for all x in this domain $\mathbb{\Gamma}(x) \leq \Gamma(x)$. If Γ_1 and Γ_2 are two typing environment, we define $\Gamma_1 \wedge \Gamma_2$ by $(\Gamma_1 \wedge \Gamma_2)(x) = \Gamma_1(x) \wedge \Gamma_2(x)$ (undefined when one of the $\Gamma_i(x)$ is not defined). Note that if $\mathbb{\Gamma} \leq \Gamma_1$ and $\mathbb{\Gamma} \leq \Gamma_2$, then $\mathbb{\Gamma} \leq \Gamma_1 \wedge \Gamma_2$.

Lemma 60 (Correctness) *If $\mathbb{\Gamma}[e] \leq t$, then there exists $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e : t$.*

Proof: By induction over the structure of e.
$\underline{e = c}$. We have $b_c \leq t$, and thus $\vdash c : t$. We can take for Γ an arbitrary typing environment such that $\Gamma \geq \mathbb{\Gamma}$. We use the $\wedge$ operator on typing environment and Lemma 30 to reconcile different Γ's given by several uses of the induction hypothesis.
$\underline{e = x}$. We have $\Gamma(x) \leq t$. We can choose $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma(x) = t$.
$\underline{e = (e_1, e_2)}$. We have $\mathbb{\Gamma}[e_1] \otimes \mathbb{\Gamma}[e_2] \leq t$. We can thus find $t_1 \geq \mathbb{\Gamma}[e_1]$ and $t_2 \geq \mathbb{\Gamma}[e_2]$ such that $t_1 \times t_2 \leq t$. The induction hypothesis gives $\Gamma_1 \geq \mathbb{\Gamma}$ such that $\Gamma_1 \vdash e_1 : t_1$ and $\Gamma_2 \geq \mathbb{\Gamma}$ such that $\Gamma_2 \vdash e_2 : t_2$. We take $\Gamma = \Gamma_1 \wedge \Gamma_2$.
$\underline{e = e_1 e_2}$. We have $\mathbb{\Gamma}[e_1] \bullet \mathbb{\Gamma}[e_2] \leq t$. We can thus find t_1, t_2 such that $t_1 \rightarrow t_2 \geq \mathbb{\Gamma}[e_1]$, $t_1 \geq \mathbb{\Gamma}[e_2]$ and $t_2 \leq t$. The induction hypothesis gives $\Gamma_1 \geq \mathbb{\Gamma}$ such that $\Gamma_1 \vdash e_1 : t_1 \rightarrow t_2$ and $\Gamma_2 \geq \mathbb{\Gamma}$ such that $\Gamma_2 \vdash e_2 : t_1$. We take $\Gamma = \Gamma_1 \wedge \Gamma_2$.
$\underline{e = \pi_i(e')}$. We have $\pi_i(\mathbb{\Gamma}[e']) \leq t$. We can thus find t_1, t_2 such that $t_1 \times t_2 \geq \mathbb{\Gamma}[e']$ and $t_i \geq t$. The induction hypothesis gives $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e' : t_1 \times t_2$. We deduce that $\Gamma \vdash e : t_i$ and by subsumption $\Gamma \vdash e : t$.

$e = (x = e' \in t' \ ?\ e_1 \mid e_2)$. We take $\mathbb{t}_0 = \mathbb{\Gamma}[e']$, $\mathbb{t}_1 = t' \oslash \mathbb{t}_0$ and $\mathbb{t}_2 = (\neg t') \oslash \mathbb{t}_0$. We also take $\mathbb{s}_1$ and $\mathbb{s}_2$ as in the corresponding case of the definition of $\mathbb{\Gamma}[e]$. We have $\mathbb{s}_1 \oslash \mathbb{s}_2 \leq t$. We can thus find $s_1 \geq \mathbb{s}_1$ and $s_2 \geq \mathbb{s}_2$ such that $t \geq s_1 \vee s_2$. Let's take $i \in \{1, 2\}$. We will define a type t_i. We have $\mathbb{s}_i \neq \Omega$ since $s_i \geq \mathbb{s}_i$. There remains two cases. If $\mathbb{t}_i \not\leq \mathbb{0}$, we have $\mathbb{s}_i = ((x : \mathbb{t}_i), \mathbb{\Gamma})[e_i]$. The induction hypothesis gives $\Gamma_i \geq \mathbb{\Gamma}$ and $t_i \geq \mathbb{t}_i$ such that $(x : t_i), \Gamma_i \vdash e_i : s_i$. Otherwise, we have $\mathbb{s}_i = \mathbb{0}$ and we take $t_i = \mathbb{0}$. In both case, we have $t_i \geq \mathbb{t}_i$. Let's consider the type $t_0 = (t_1 \wedge t') \vee (t_2 \wedge \neg t')$. We now prove that $t_0 \geq \mathbb{t}_0$. Since $t_1 \geq \mathbb{t}_1 = t' \oslash \mathbb{t}_0$, there exists $t'_1 \geq \mathbb{t}_0$ such that $t' \wedge t'_1 \leq t_1$. Similarly, we have $t'_2 \geq \mathbb{t}_0$ such that $(\neg t') \wedge t'_2 \leq t_2$. We get $t_0 \geq (t' \wedge t'_1) \vee ((\neg t') \wedge t'_2) \geq (t' \wedge t'_1 \wedge t'_2) \vee ((\neg t') \wedge t'_1 \wedge t'_2) \simeq t'_1 \wedge t'_2 \geq \mathbb{t}_0$.
Since $t_0 \geq \mathbb{t}_0$, the induction hypothesis gives $\Gamma_0 \geq \mathbb{\Gamma}$ such that $\Gamma_0 \vdash e' : t_0$. Let's consider the types $t''_1 = t_0 \wedge t \leq t_1$ and $t''_2 = t_0 \wedge (\neg t) \leq t_2$. By considering the intersection of Γ_0 and of Γ_1 and Γ_2 when they are defined, we find $\Gamma \geq \mathbb{\Gamma}$ such that $\Gamma \vdash e' : t_0$ and $(x_i : t''_i), \Gamma \vdash e_i : s_i$ when $\mathbb{t}_i \not\leq \mathbb{0}$. The rule (*case*) gives $\Gamma \vdash e : s_1 \vee s_2$. By subsumption, we get $\Gamma \vdash e : t$.
$e = \mu f(t_1 \rightarrow s_1; \ldots; t_n \rightarrow s_n).\lambda x.e'$. We take $\mathbb{t}$ and $\mathbb{s}_i$ as in the definition of the corresponding case for $\mathbb{\Gamma}[e]$. Since $\mathbb{\Gamma}[e] \neq \Omega$, we get $\mathbb{t} \leq t$ and $\mathbb{s}_i \leq s_i$ for all $i = 1..n$. The induction hypothesis gives, for each i, an environment $\Gamma_i \geq \mathbb{\Gamma}$, and two types $t^i \geq \mathbb{t}$, $t''_i \geq t_i$ such that $(f : t^i), (x : t''_i), \Gamma_i \vdash e' : s_i$.
We define the type t' as $\bigwedge_{i=1..n} t^i \wedge t$. We have $t' \geq \mathbb{t} = [t_i \rightarrow s_i]_{i=1..n}$. We can thus find a type t'' of the form $t'' = \bigwedge_{i=1..n} t_i \rightarrow s_i \wedge \bigwedge_{j=1..m} \neg(t'_j \rightarrow s'_j)$ such that $t' \geq t''$ and $t'' \not\simeq \mathbb{0}$.
If we take for Γ the intersection of all the Γ_i, we obtain $(f : t''), (x : t_i), \Gamma \vdash e' : s_i$ for all i from which we conclude $\Gamma \vdash e : t''$ and thus $\Gamma \vdash e : t$. □

Lemma 61 (Completeness) *If* $\mathbb{\Gamma} \leq \Gamma$ *and* $\Gamma \vdash e : t$ *then* $\mathbb{\Gamma}[e] \leq t$.

Proof: By induction over the derivation of $\Gamma \vdash e : t$ and case disjunction over the last rule used in this derivation. The proof is mechanical. We give the details only for the rule (*case*).

$$\frac{\Gamma \vdash e : t_0 \qquad \left\{ \begin{array}{lcl} t_0 \not\leq \neg t & \Rightarrow & (x : t_0 \wedge t), \Gamma \vdash e_1 : s \\ t_0 \not\leq t & \Rightarrow & (x : t_0 \backslash t), \Gamma \vdash e_2 : s \end{array} \right.}{\Gamma \vdash (x = e \in t \ ?\ e_1 | e_2) : s}$$

We assume that $\mathbb{\Gamma} \leq \Gamma$ and we take $\mathbb{t}_0$,$\mathbb{t}_1$, $\mathbb{t}_2$,$\mathbb{s}_1$,$\mathbb{s}_2$ as in the definition of $\mathbb{\Gamma}[(x = e \in t \ ?\ e_1 | e_2)]$. We need to prove that $\mathbb{s}_1 \oslash \mathbb{s}_2 \leq s$, that is $\mathbb{s}_1 \leq s$ and $\mathbb{s}_2 \leq s$. We will do the proof for $\mathbb{s}_1$ (the proof for $\mathbb{s}_2$ is similar).
The induction hypothesis gives $\mathbb{t}_0 = \mathbb{\Gamma}[e] \leq t_0$, from which we get $\mathbb{t}_1 \leq t \wedge t_0$. If $\mathbb{t}_1 \leq \mathbb{0}$, then $\mathbb{s}_1 = \mathbb{0} \leq s$. Otherwise, since $\{\mathbb{t}_1\} \neq \varnothing$, we have $\mathbb{s}_1 = ((x : \mathbb{t}_1), \mathbb{\Gamma})[e_1]$. We have $t_0 \not\leq \neg t$, otherwise $\mathbb{t}_1 \leq \mathbb{0}$. We thus have a sub-derivation $(x : t_0 \wedge t), \Gamma \vdash e_1 : s$. The induction hypothesis, applied to the environment $(x : \mathbb{t}_1), \mathbb{\Gamma}$ gives $\mathbb{s}_1 \leq s$. □

By combining the two previous lemmas, we get an exact characterization of the type-checking algorithm in terms of the type system.

Theorem 62 *For any scheme environment $\mathbb{\Gamma}$ and expression e:*

$$\mathbf{\{}\mathbb{\Gamma}[e]\mathbf{\}} = \{t \mid \exists \Gamma \geq \mathbb{\Gamma}.\Gamma \vdash e : t\}$$

Corollary 63 *Let Γ be a typing environment. It can also be seen as a scheme environment. For any expression e and any type t, we have:*

$$\Gamma \vdash e : t \iff \Gamma[e] \leq t$$

As a special case, the expression e is well-typed under Γ if and only if $\mathbf{\{}\Gamma[e]\mathbf{\}} \neq \varnothing$.

7 Conclusion

Our original motivation for developing the theory presented in this article was the addition of first-class functions to XDuce while preserving the set-theoretic approach to subtyping. This was the starting point of the $\mathbb{C}$Duce project [10], aiming at developing a programming framework covering several aspects of XML programming: efficient implementation, query languages, web-services, web programming, and so on.

The reader might be surprised to face such a complex theory in the setting of an XML-oriented functional language. First, we should mention that XML plays no role in the complexity of the theory. The circularity which our bootstrapping technique addresses comes only from the combination of arrow types, recursive types and Boolean connectives. Since XDuce already had recursive types and Boolean connectives, it seemed natural to add arrow types and to fully integrate them with these features. Simpler solutions could have been possible, e.g. by stratifying the type algebra so as to avoid any interaction between arrow types and existing XDuce types: this is what the first author did to integrate XDuce types into an ML-based type system [15].

Second, we could have presented the theory without introducing the abstract concept of models. Indeed, for the application to a specific programming language, we could have worked directly with the universal model (Section 6.8). That said, we believe that the current presentation better captures the essence of our approach. Working directly with a specific model would be mysterious and ad hoc.

Although we presented our notion of model and the bootstrapping technique on a specific type algebra and for a specific calculus, our framework is quite robust. Frisch's Ph.D. thesis [14] describes some variants of the system (removing type error at application, removing overloading) and shows how minimal modifications to the theory are enough to deal with them.

More importantly, our approach and the techniques we developed turned out to have much a broader application than we initally expected. What we devised is the first approach for a higher order λ-calculus in which union, intersection, *and* negation types have a set-theoretic interpretation. The logical

relevance of the approach was independently confirmed by Dezani *et al.* [13] who showed that the subtyping relation induced by the universal model of Section 6.8 restricted to its positive part (that is arrows, unions, intersections but no negations) coincides with the relevant entailment of the $\mathbf{B}_+$ logic (defined 30 years before we started our work). This same approach can be applied to paradigms other than λ-calculi: Castagna, De Nicola and Varacca [9] use our technique to define the $\mathbb{C}\pi$-calculus, a π-calculus where Boolean combinators are added to the type constructors $\mathtt{ch}^+(t)$ and $\mathtt{ch}^-(t)$ which classify all the channels on which it is possible to read or, respectively, to write a value of type t. The technique using the extensional interpretation is still needed for cardinality reasons, however bootstrapping in $\mathbb{C}\pi$ has a different flavour, since it generates a model that is much closer to the model of values. Interestingly, this model is defined by a fix-point construction. $\mathbb{C}\pi$ features several points that are in common with or dual to $\mathbb{C}$Duce: $\mathbb{C}\pi$ presents the same paradox one meets when adding reference types to $\mathbb{C}$Duce [7]. The paradox can be avoided by restricting $\mathbb{C}\pi$ to its "local" version [9] but in that case the type schemes of Section 6.12 must be reintroduced, in spite of the fact that they are not needed for the full version of $\mathbb{C}\pi$. Another striking resemblance between $\mathbb{C}$Duce and $\mathbb{C}\pi$ that is worth mentioning is that in order to decide the subtyping relation for the $\mathbb{C}\pi$, one tackles the same difficulties as those met in deciding general subtyping for the polymorphic extension of $\mathbb{C}$Duce [19], namely, one must be able to decide whether a type is a singleton or not. An informal introduction to these aspects can be found in [5], while the formal correspondence between $\mathbb{C}$Duce and $\mathbb{C}\pi$ is studied in [6].

References

[1] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of the Seventh ACM Conference on Functional Programming and Computer Architecture*, pages 31–41, Copenhagen, Denmark, June 93.

[2] A. Asperti and G. Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT-Press, 1991.

[3] H. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *Journal of Symbolic Logic*, 48(4):931–940, 1983.

[4] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-friendly general purpose language. In *ICFP '03, 8th ACM International Conference on Functional Programming*, pages 51–63, Uppsala, Sweden, 2003. ACM Press.

[5] G. Castagna. Semantic subtyping: challenges, perspectives, and open problems. In *ICTCS 2005, Italian Conference on Theoretical Computer Science*,

number 3701 in Lecture Notes in Computer Science, pages 1–20. Springer, 2005.

[6] G. Castagna, M. Dezani, and D. Varacca. Encoding $\mathbb{C}$Duce into the $\mathbb{C}\pi$-calculus. In *CONCUR '06*, Lecture Notes in Computer Science. Springer-Verlag, 2006. To appear.

[7] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of *PPDP '05, the 7th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming,* ACM Press (full version) and *ICALP '05, 32nd International Colloquium on Automata, Languages and Programming,* Lecture Notes in Computer Science n. 3580, Springer (summary), Lisboa, Portugal, 2005. Joint ICALP-PPDP keynote talk.

[8] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.

[9] G. Castagna, R. D. Nicola, and D. Varacca. Semantic subtyping for the π-calculus. In *LICS '05, 20th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 2005.

[10] The $\mathbb{C}$Duce programming language. `http://www.cduce.org`.

[11] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre-Dame Journal of Formal Logic*, 21(4):685–693, October 1980.

[12] F. Damm. Subtyping with union types, intersection types and recursive types II. Research Report 816, IRISA, 1994.

[13] M. Dezani-Ciancaglini, A. Frisch, E. Giovannetti, and Y. Motohama. The relevance of semantic subtyping. In *Intersection Types and Related Systems*. Electronic Notes in Theoretical Computer Science 70(1), 2002.

[14] A. Frisch. *Théorie, conception et réalisation d'un langage de programmation fonctionnel adapté à XML.* PhD thesis, Université Paris 7, Dec. 2004.

[15] A. Frisch. OCaml + XDuce. In *Programming Languages Technologies for XML (PLAN-X)*, 2006.

[16] A. Frisch, G. Castagna, and V. Benzaken. Semantic Subtyping. In *LICS '02, 17th Annual IEEE Symposium on Logic in Computer Science*, pages 137–146. IEEE Computer Society Press, 2002.

[17] R. Hindley and G. Longo. Lambda-calculus models and extensionality. *Zeit. Math. Logik Grund. Math.*, 26(2):289–319, 1980.

[18] H. Hosoya. *Regular Expression Types for XML.* PhD thesis, The University of Tokyo, 2001.

[19] H. Hosoya, A. Frisch, and G. Castagna. Parametric polymorphism for XML. In *POPL '05, 32nd ACM Symposium on Principles of Programming Languages*. ACM Press, 2005.

[20] H. Hosoya and B. Pierce. Regular expression pattern matching for XML. In *POPL '01, 25th ACM Symposium on Principles of Programming Languages*, 2001.

[21] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[22] J. C. Reynolds. The coherence of languages with intersection types. In T. Ito and A. R. Meyer, editors, *Theoretical Aspects of Computer Software*, volume 526 of *Lecture Notes in Computer Science*, pages 675–700, Berlin, 1991. Springer-Verlag.

[23] J. C. Reynolds. Design of the programming language Forsythe. Technical Report CMU-CS96 -146, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1996.